SSCM Exercise 2

Nikolaus Czernin - 11721138

```
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
# install.packages("FRACTION")
library("FRACTION")
```

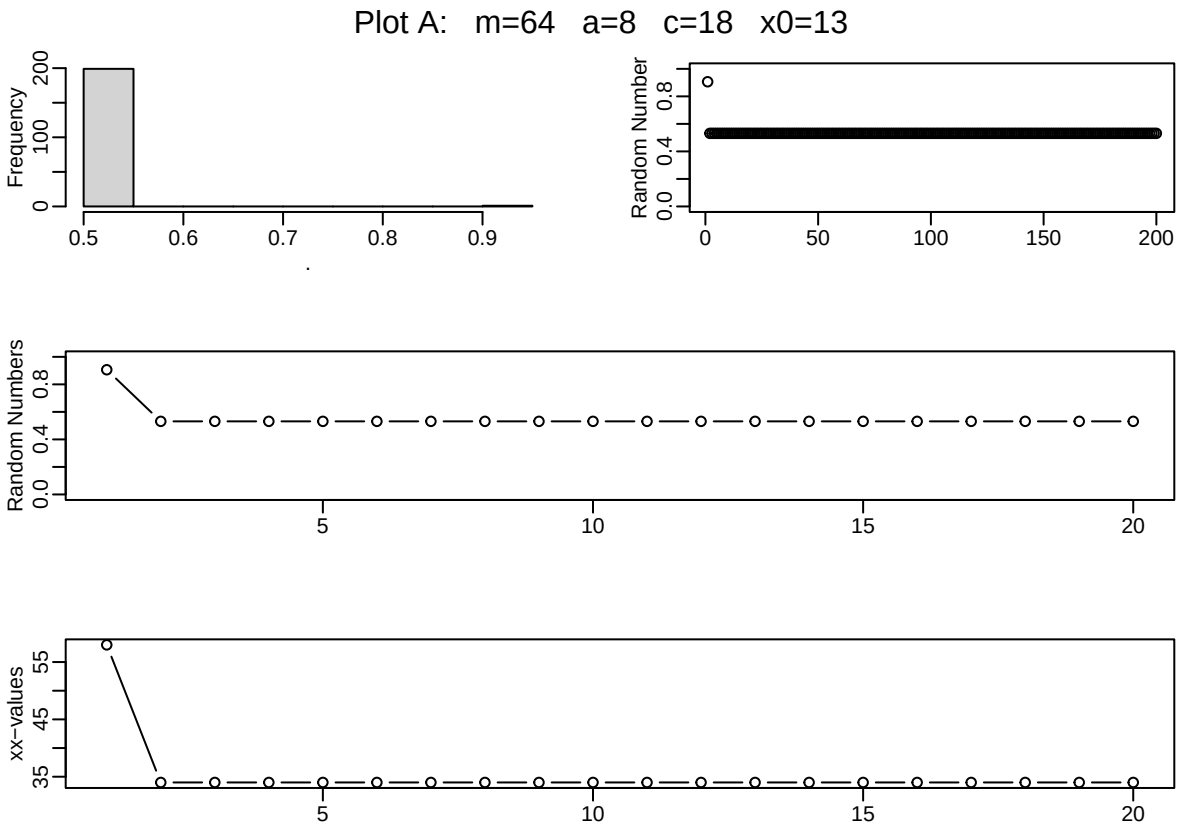
Linear Congruential Random Number Generator

```
# prepare the PRNG function
lcrng <- function(n, m, a, c=0, x0){
  us <- numeric(n)
  # keep an extra vector for the x values
  xs <- numeric(n)
  for (i in 1:n){
    x0 <- (a * x0 + c) %% m
    xs[i] <- x0
    us[i] <- x0 / m
  }
  list("u"=us, "x"=xs)
}


visualize_random_numbers <- function(n, m, a, c, x0, title=""){
  # generate random numbers with given params
  prns <- lcrng(n, m, a, c, x0)
  df <- data.frame(
    i = 1:n,
    u = prns$u,
    x = prns$x,
    n = n,
```

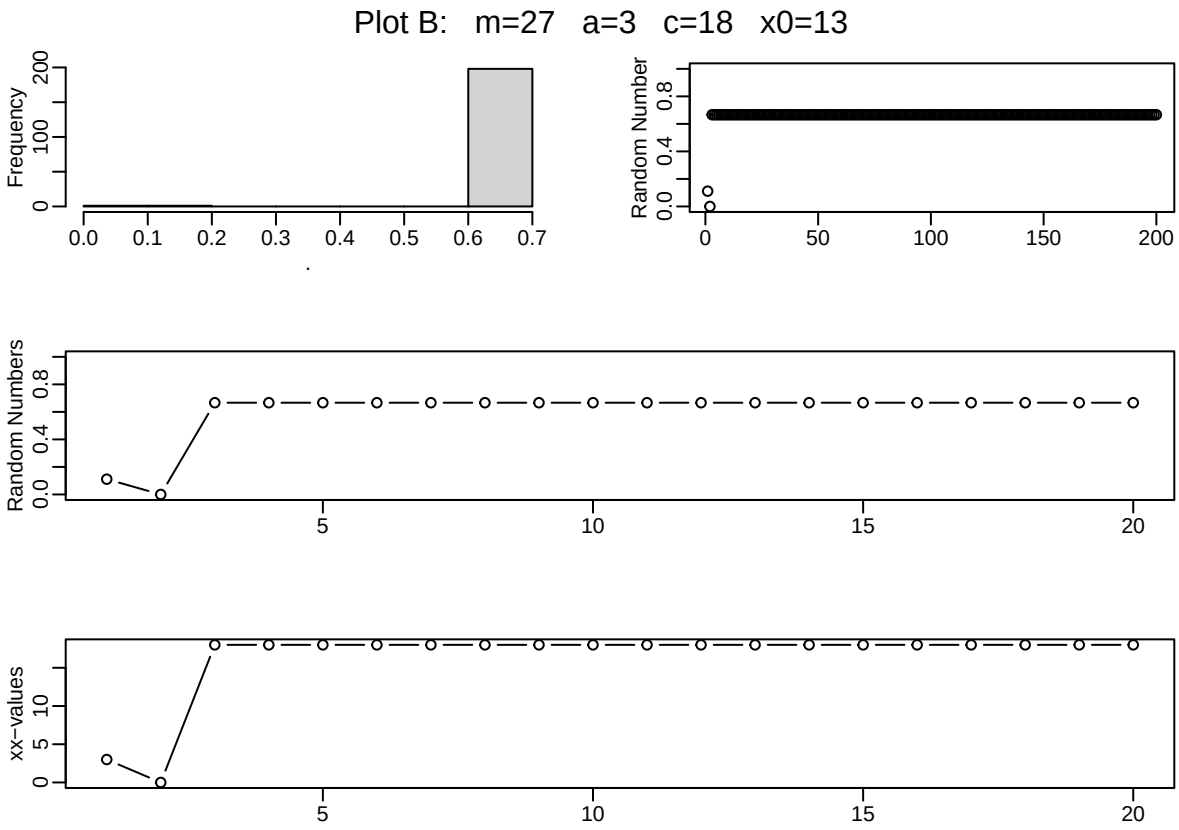
```
    m = m,
    a = a,
    c = c,
    x0 = x0
  )

  # prepare a double-plot window
  # set up the layout with 2 rows and 2 columns
  layout(matrix(c(1, 2, 3, 3, 4, 4), nrow = 3, byrow = TRUE))

  par(mar = c(3, 3.5, 2.5, 2),
      mgp = c(1.5, 0.5, 0))

  # create the plots
  df$u %>% hist(main="")
  plot(df$i, df$u, xlab="", ylab="Random Number", ylim=c(0, 1))

  # for the line plot, only use the first 20 numbers
  df_head <- df %>% head(20)
  plot(df_head$i, df_head$u, type="b", xlab="", ylab="Random Numbers", ylim=c(0, 1))
  first_cycle <- filter(df, x==x0) %>% head(1) %>% .$i
  abline(v=first_cycle, col="blue")

  # plot also the x-values
  plot(df_head$i, df_head$x, type="b", xlab="", ylab="xx-values")
  abline(v=first_cycle, col="blue")

  # make a custom title with the parameters
  text <- paste0(title, ":   m=", m, "   a=", a, "   c=", c, "   x0=", x0, "\n")
  mtext(text, side=3, outer=TRUE, line=-3)

}

# visualize_random_numbers(200, 9, 8, 7, 6, "Plot A")
```

In this chunk, I defined a PRNG function that uses the Linear Congruential Random Number Generator algorithm. It expects all parameters as arguments. I defined another function that visualizes the result using a histogram, a scatterplot and a lineplot of the first values to visualize a sequence-loop. It also visualizes the x-values for better analysis.

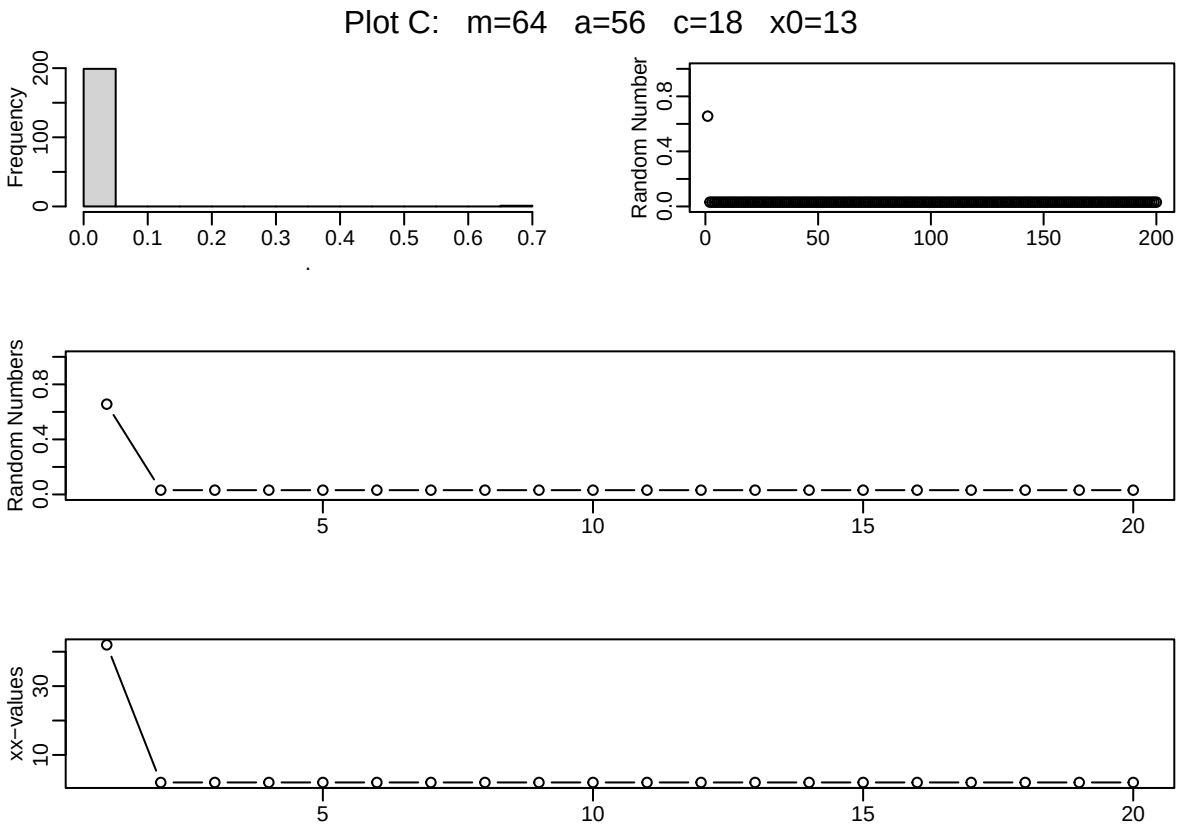
```
visualize_random_numbers(200,  64,   8,  18,  13, "Plot A")
```



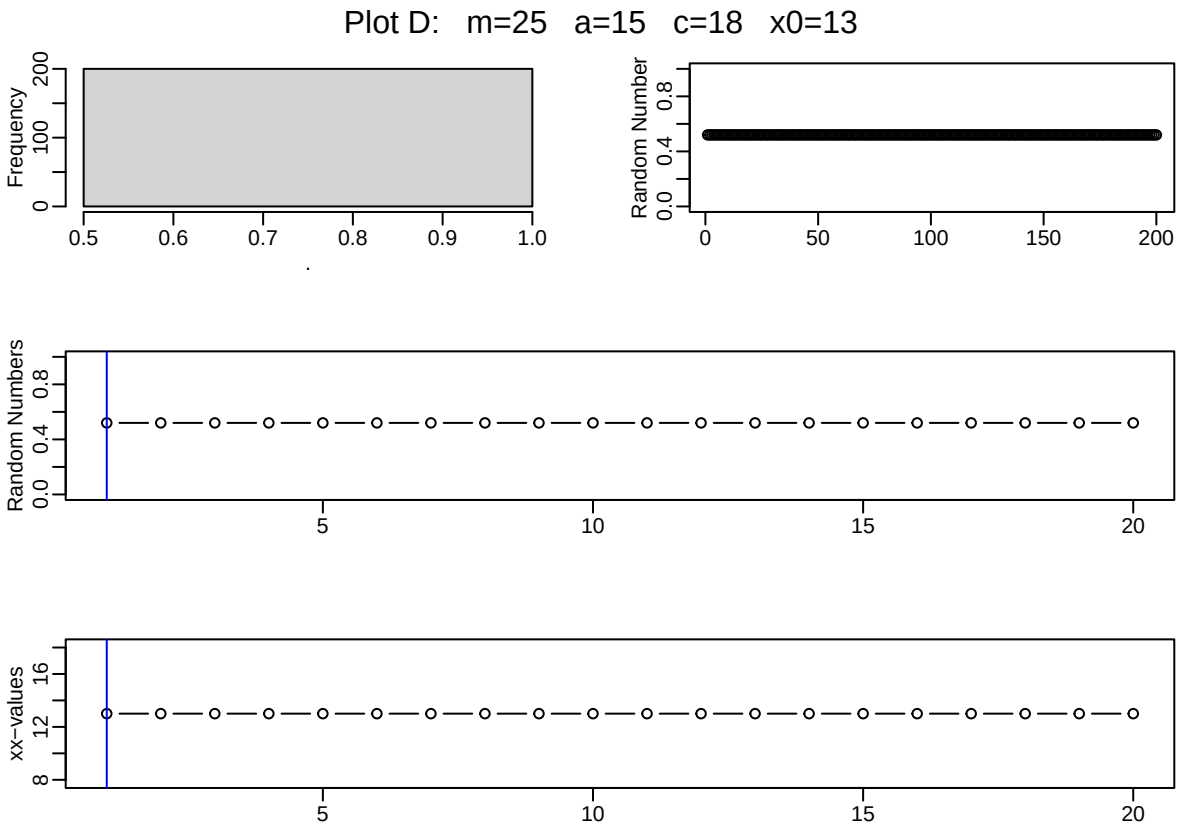

```
visualize_random_numbers(200,  27,   3,  18,  13, "Plot B")
```




```
visualize_random_numbers(200,  64,  56,  18,  13, "Plot C")
```



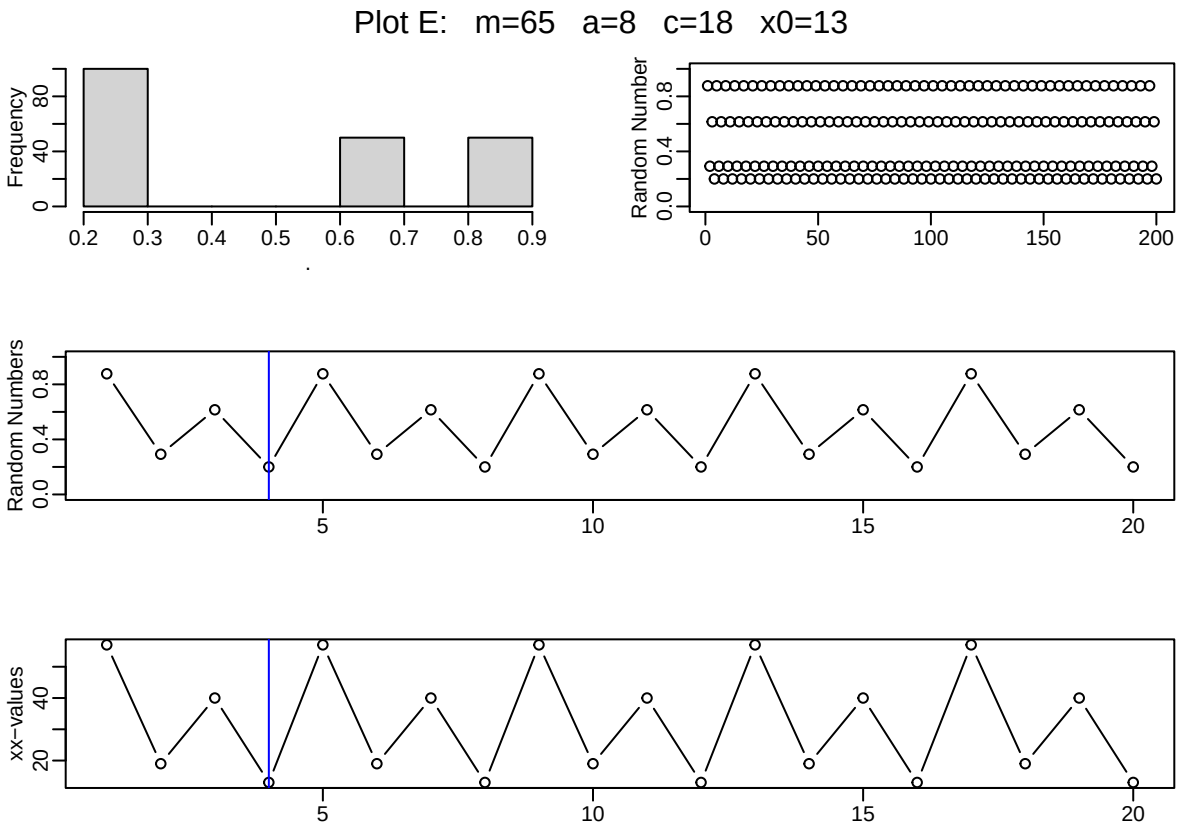

```
visualize_random_numbers(200,  25,  15,  18,  13, "Plot D")
```



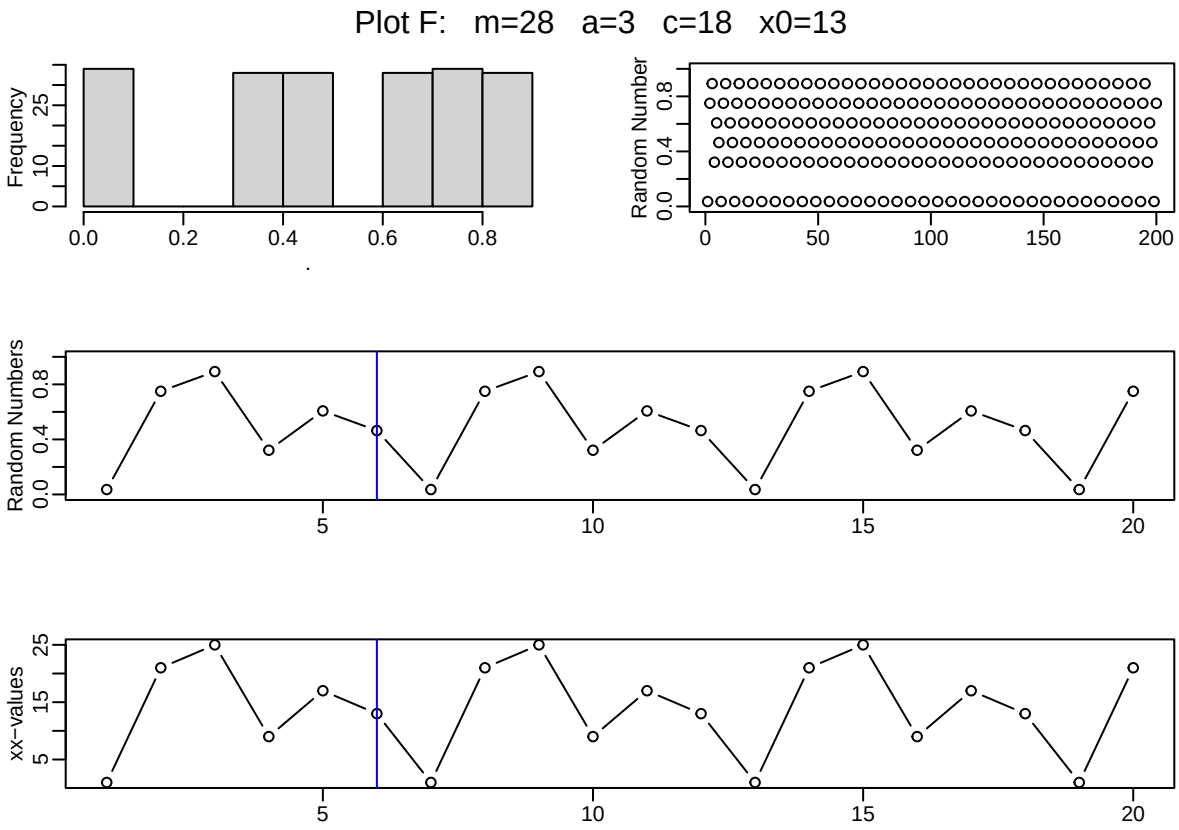
Plots A through D above all loop very early. What they have in common is that m and a have common denominators. After numbers in the sequence, the values stagnate.

The blue lines mark the first iteration, where the generated x-value is equal to the initial x_0 value, which is the latest point of a cycle restarting, though a cycle may start even earlier too.

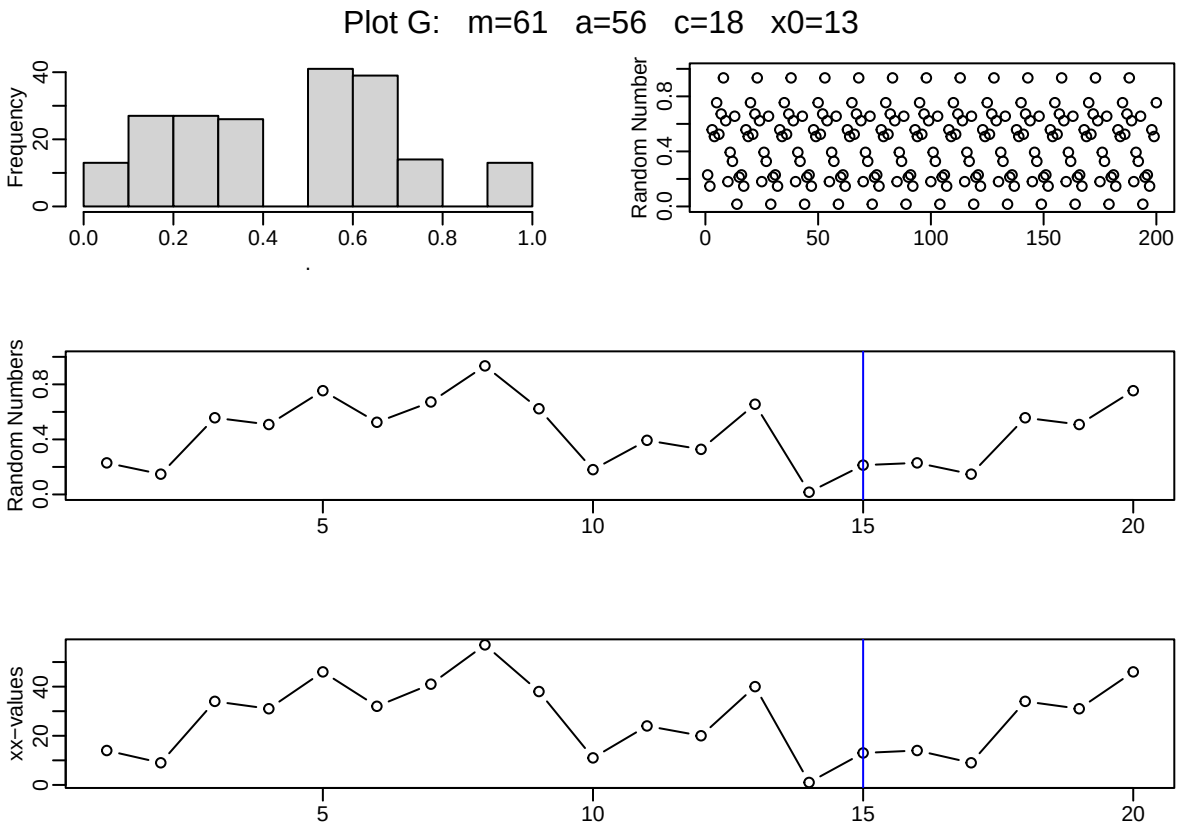
```
visualize_random_numbers(200,  65,   8,  18,  13, "Plot E")
```



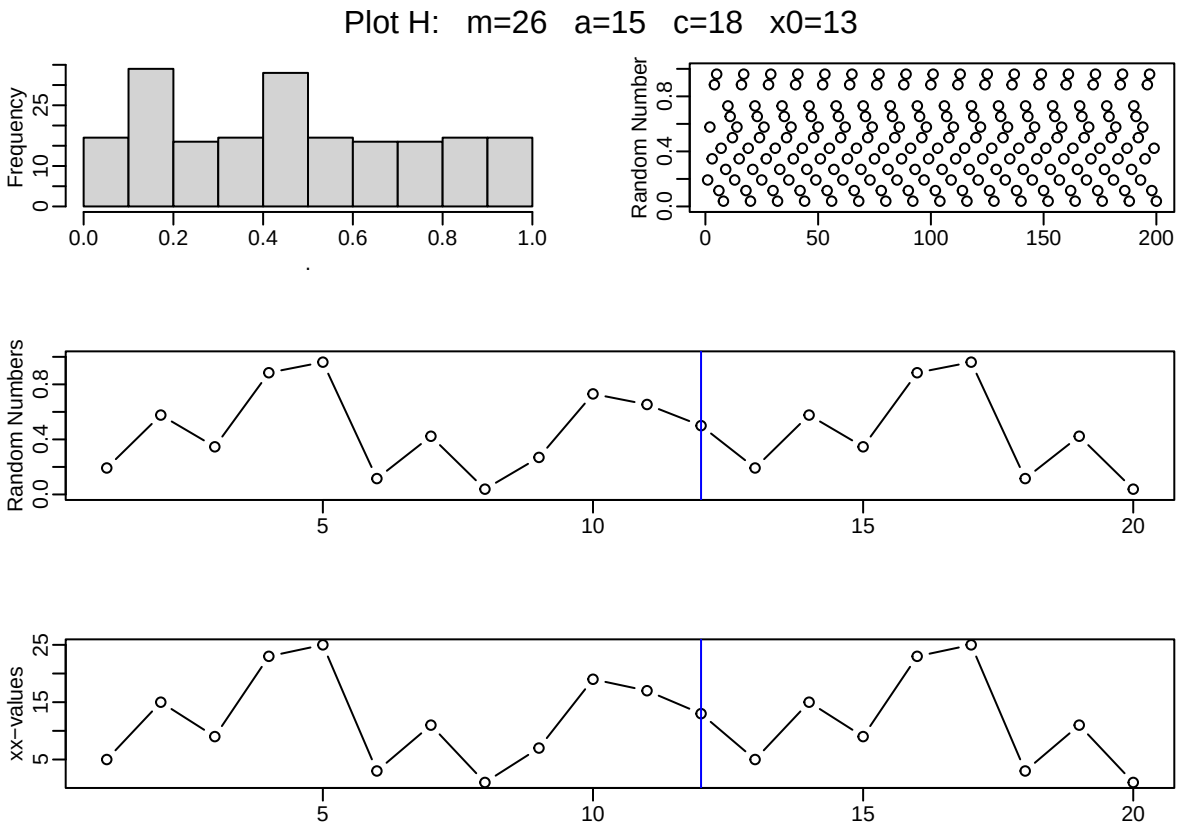

```
visualize_random_numbers(200,  28,   3,  18,  13, "Plot F")
```




```
visualize_random_numbers(200,  61,  56,  18,  13, "Plot G")
```

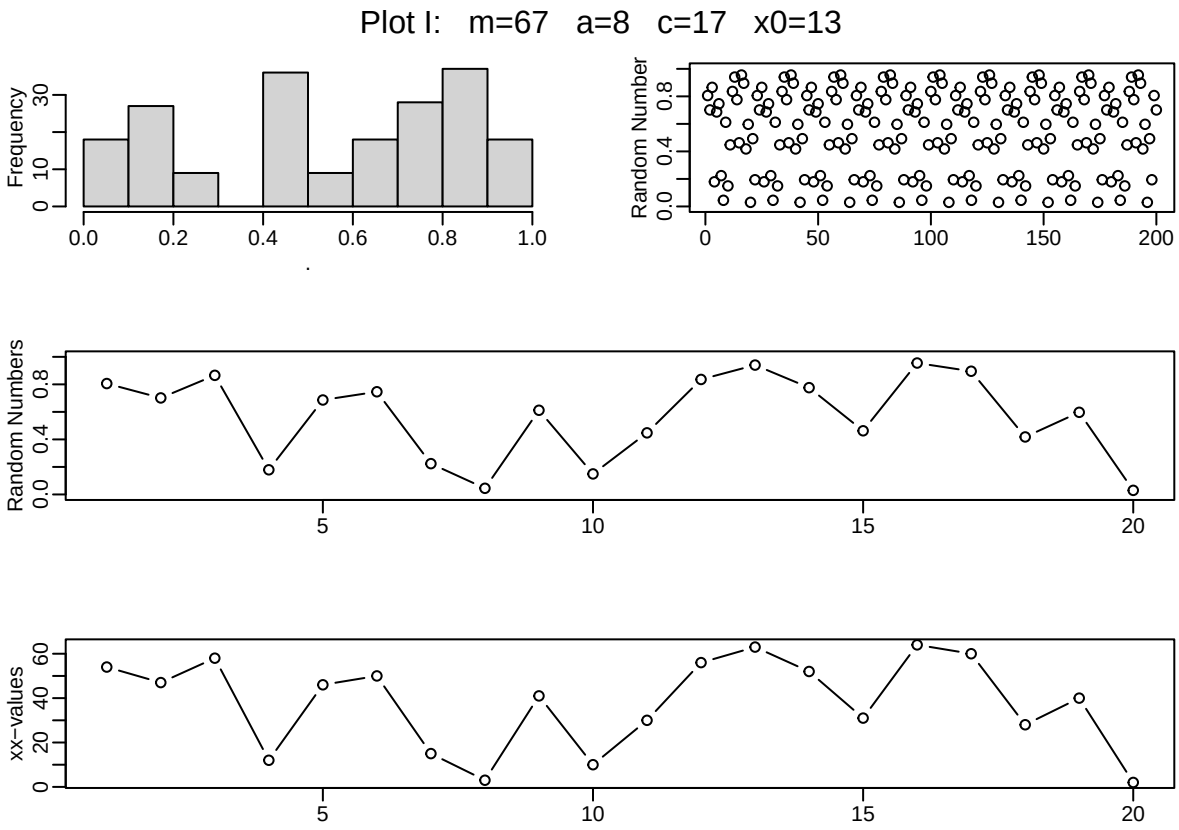



```
visualize_random_numbers(200,  26,  15,  18,  13, "Plot H")
```

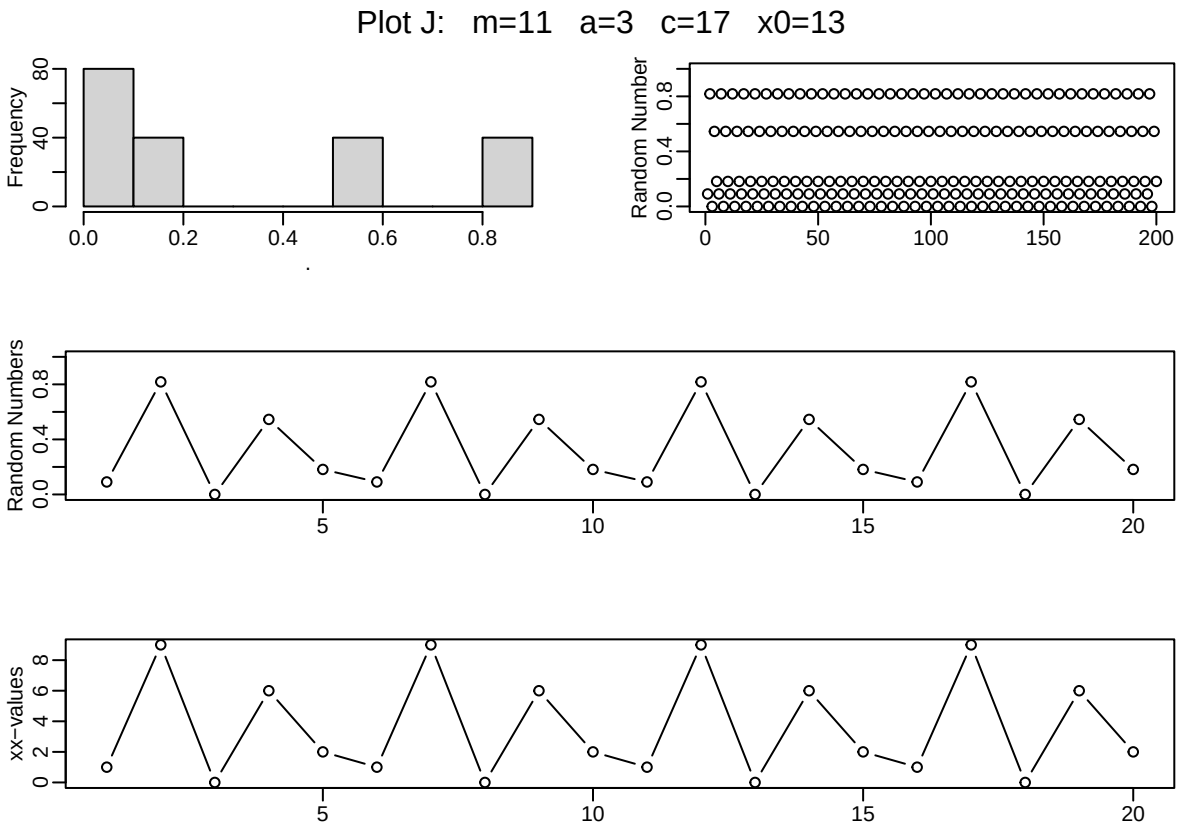


When using m values that have no common denominator with a the loops are created later, as seen in plots E through H.

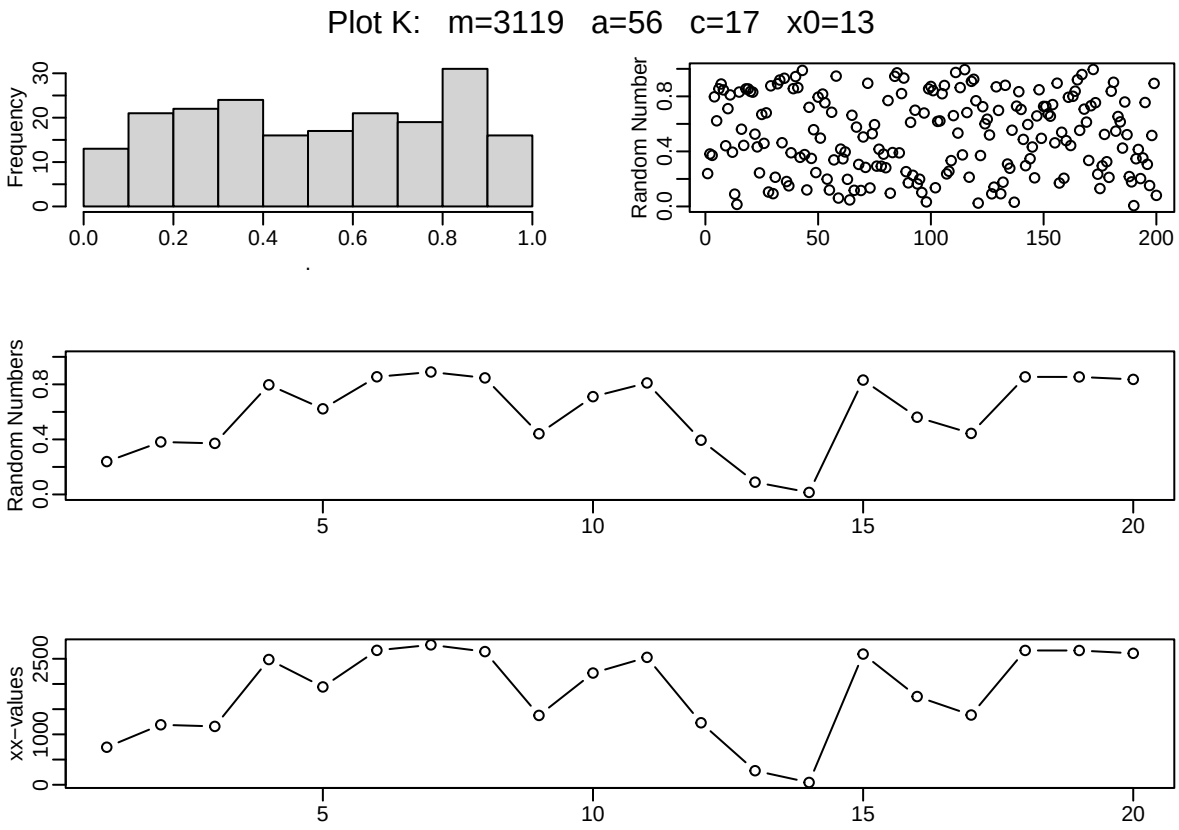
```
visualize_random_numbers(200,  67,   8,  17,  13, "Plot I")
```



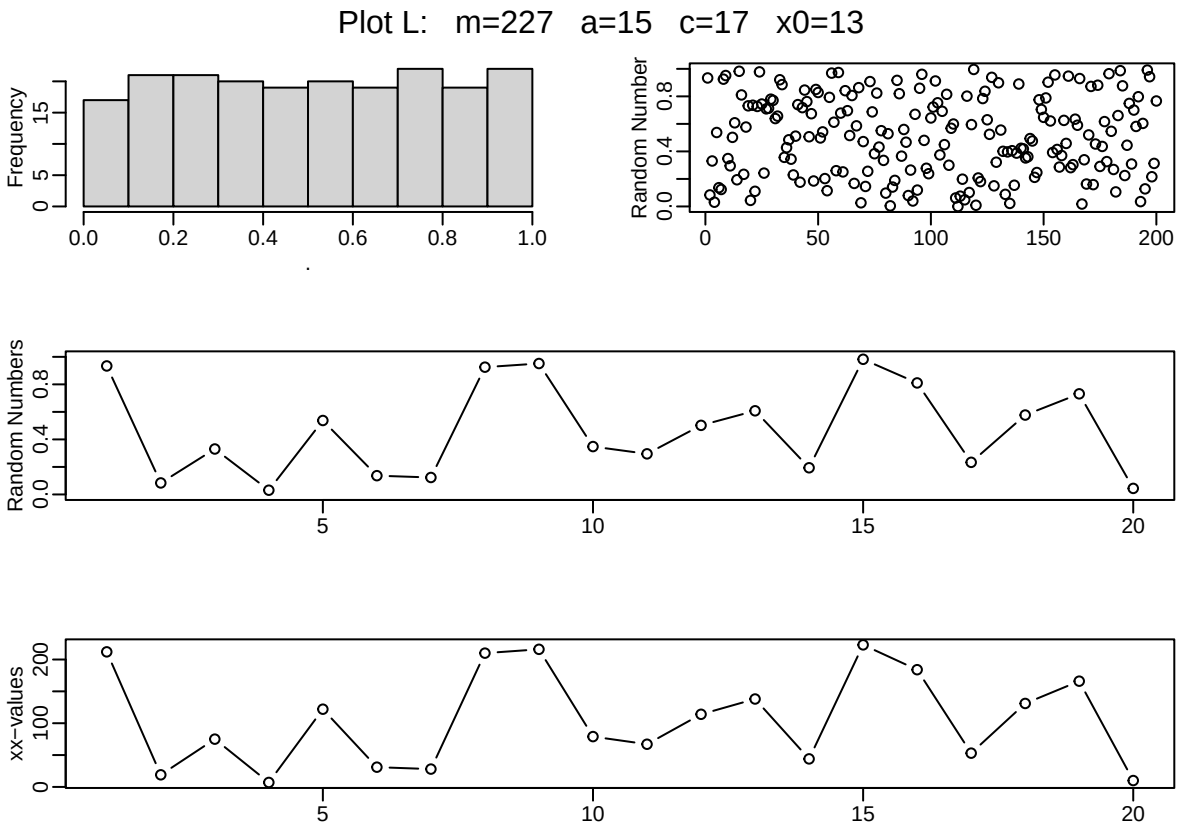

```
visualize_random_numbers(200,  11,   3,  17,  13, "Plot J")
```




```
visualize_random_numbers(200,  3119,  56,  17,  13, "Plot K")
```

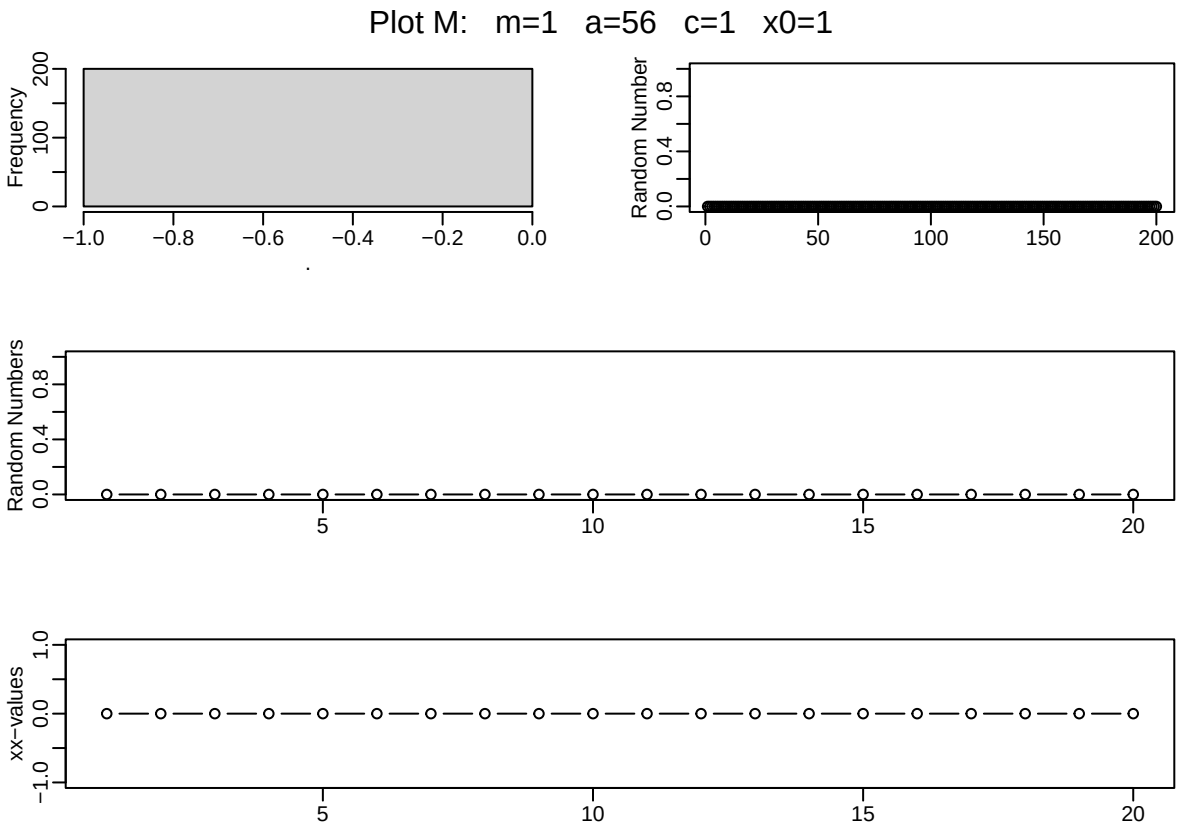



```
visualize_random_numbers(200,  227,  15,  17,  13, "Plot L")
```

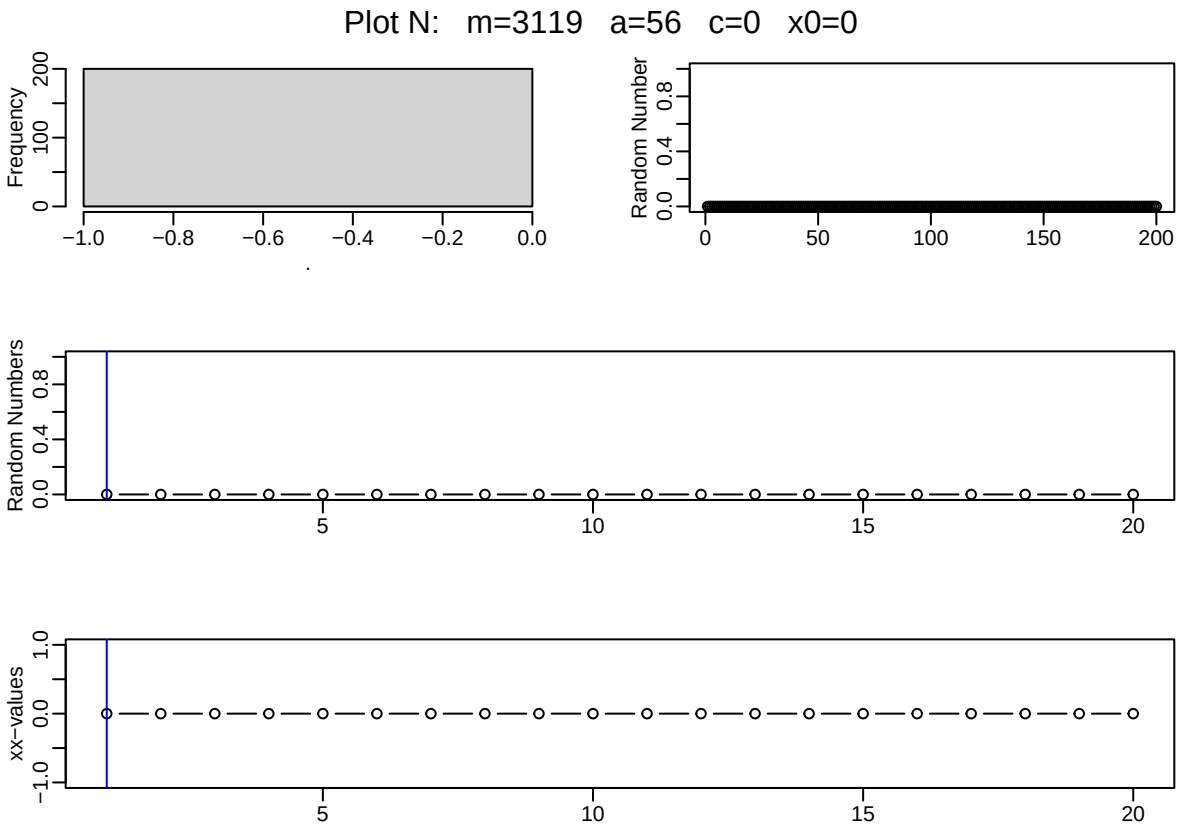


As seen in Plots I through L, when using large prime numbers as m, i.e. 3-digit value or higher, there are no apparent cycles in the random number sequences anymore.

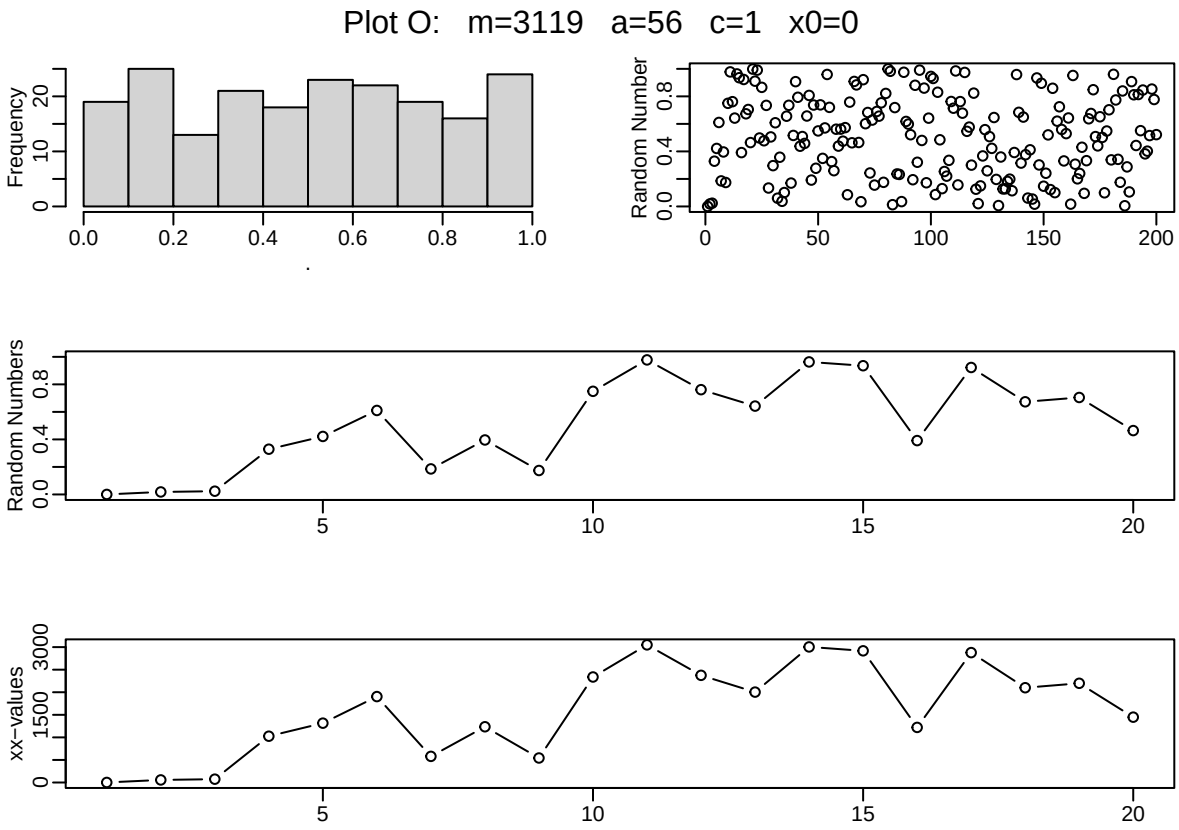
```
visualize_random_numbers(200,  1,  56,  1,  1, "Plot M")
```



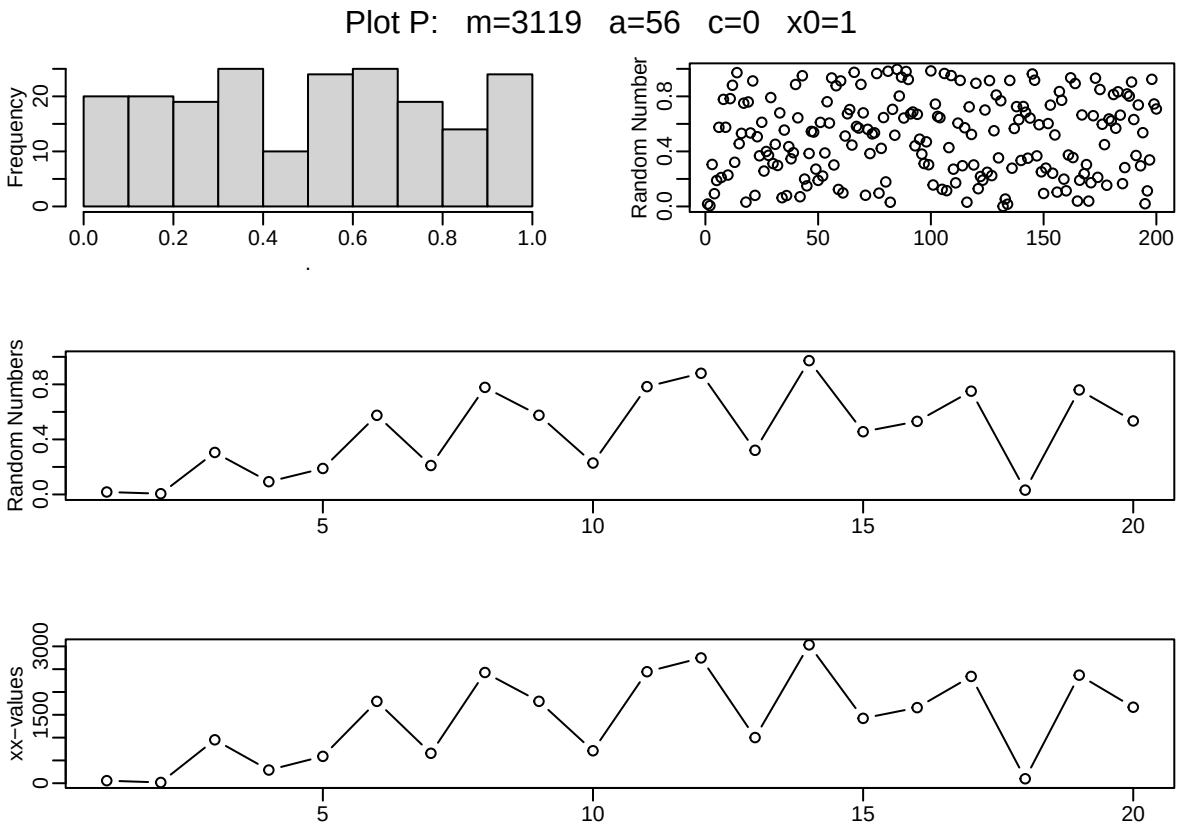

```
visualize_random_numbers(200,  3119,  56,  0,  0, "Plot N")
```




```
visualize_random_numbers(200,  3119,  56,  1,  0, "Plot 0")
```

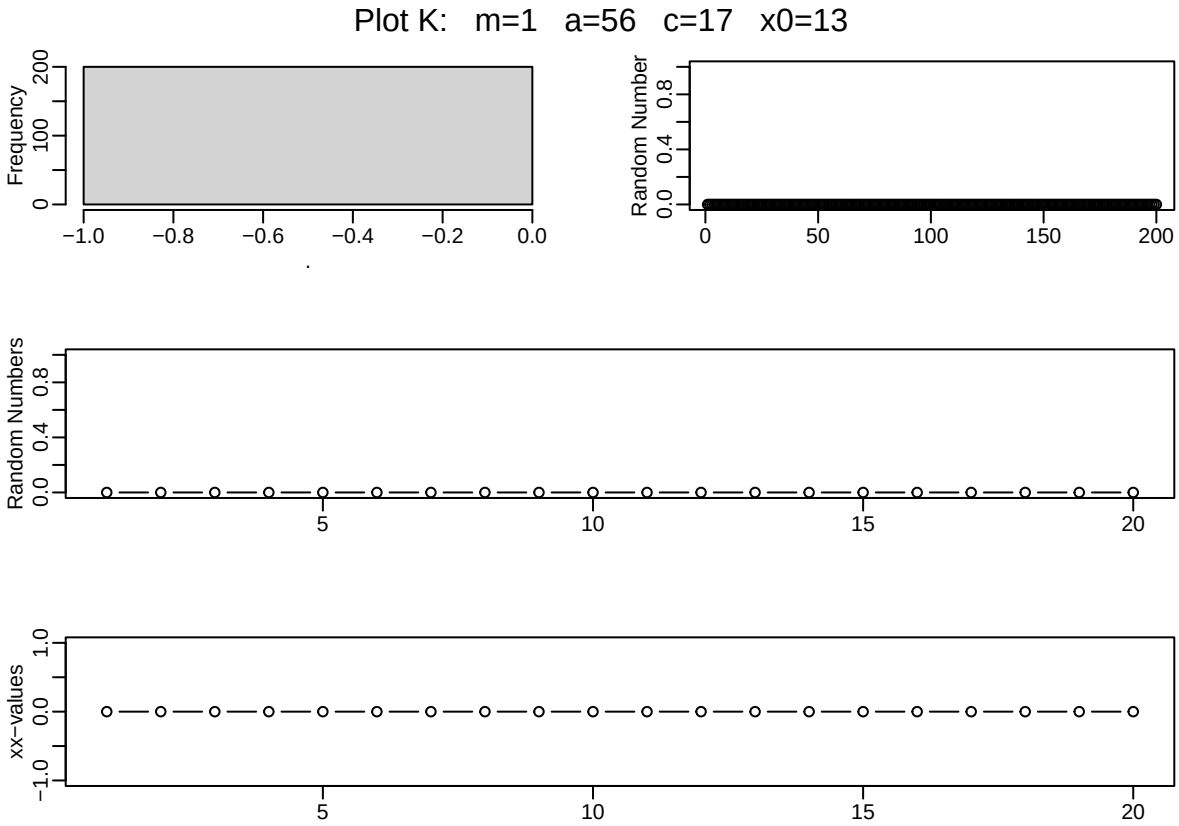



```
visualize_random_numbers(200,  3119,  56,  0,  1, "Plot P")
```



Plot M shows that if m is 1, the random numbers will be all zeros, as modulus of 1 will always be 0 on integers. Plot N shows that having both c and x_0 equal to zero leads to a sequence producing only zeros. If either are unequal to zero, you get a random sequence with seemingly no cycles, as seen in plots O and P.

```
visualize_random_numbers(200,  1,  56,  17,  13, "Plot K")
```



Sampling from the Exponential distribution

To generate a random sample from exp, you first generate a random sample from the uniform distribution = u. We then set $u = F(x)$, where $F(x)$ is the cdf of exp, also solve for x.

$$F(x) = 1 - exp(-\lambda x), \lambda > 0$$
$$1 - exp(-\lambda x) = u$$
$$1 - u = exp(-\lambda x)$$
$$ln(1-u) = -x\lambda$$
$$x = -\frac{ln(1-u)}{\lambda}$$

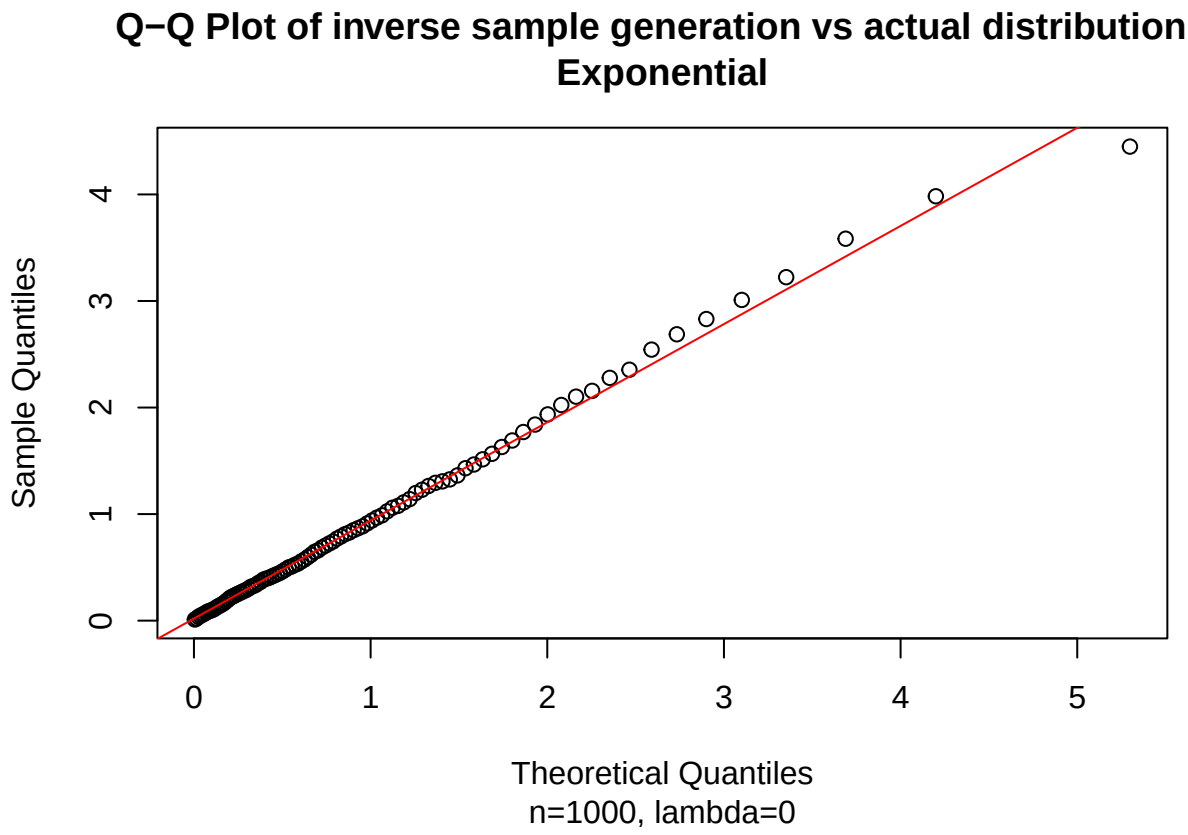
With this formula we can use a uniform variable u and convert to to a random variable x form the exponential distribution, with a parameter λ.

```
InvCdfExp <- function(x, lambda=1){
  # add a tiny number to lambda in case it is zero
  -log(1 - x) / (lambda+1e-12)
}

random_exp <- function(n, lambda=1){
  # generate n different uniform random nums
```

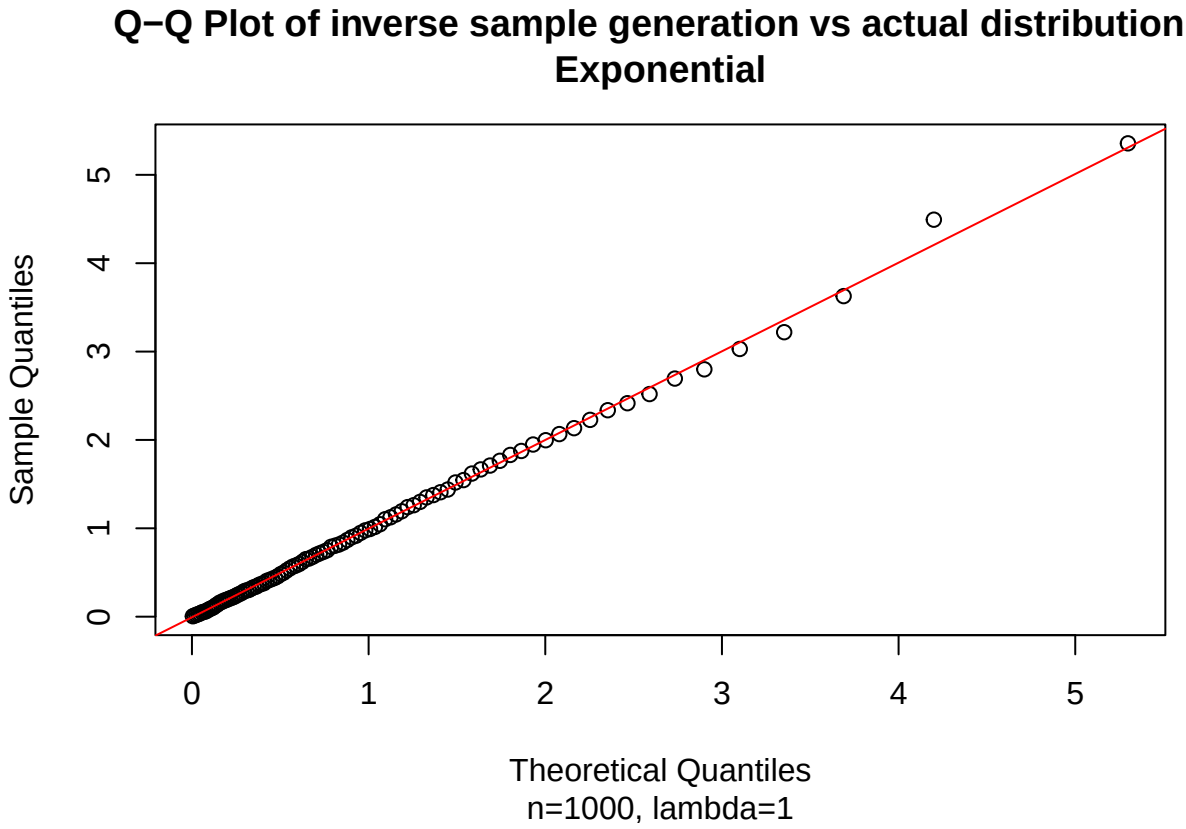
```
  us <- runif(n)
  # now apply the inverse exponential formula from above
  InvCdfExp(us)
}
```

```
plot_exp_sample <- function(x, n="", lambda=""){
  p <- ppoints(100)    # 100 equally spaced points on (0,1), excluding endpoints
  q <- quantile(x,p=p) # percentiles of the sample distribution
  plot(qexp(p), q, main="Q-Q Plot of inverse sample generation vs actual distribution: \nExponential", s
       xlab="Theoretical Quantiles", ylab="Sample Quantiles")
  qqline(q, distribution=qexp, col="red")
}
```

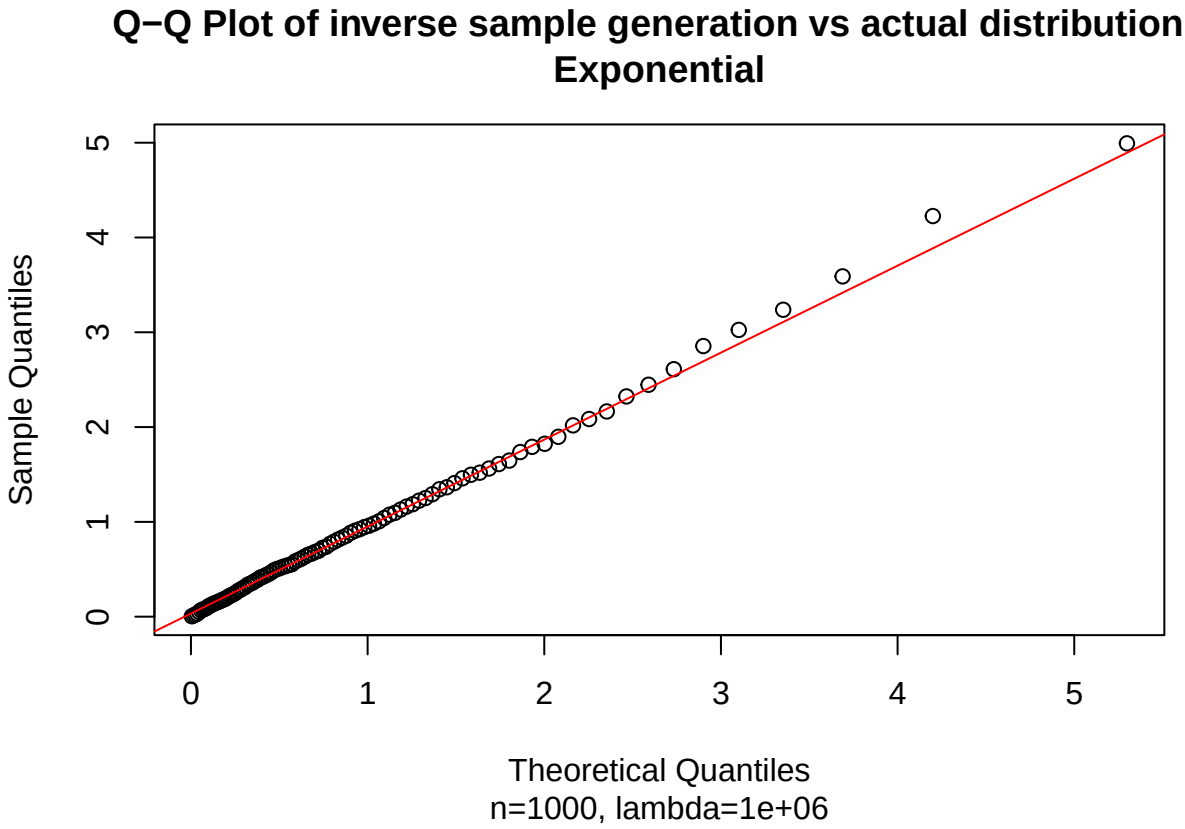
```
n <- 1000
lambda <- 0
x <- random_exp(n, lambda)
plot_exp_sample(x, n, lambda)
```




```
n <- 1000
lambda <- 1
x <- random_exp(n, lambda)
plot_exp_sample(x, n, lambda)
```




```
n <- 1000
lambda <- 1e6
x <- random_exp(n, lambda)
plot_exp_sample(x, n, lambda)
```



The random number generator generates numbers that visually match the target distribution quite closely. The the higher quantiles, there seems to be some sverving from the target distribution, but nothing grave.

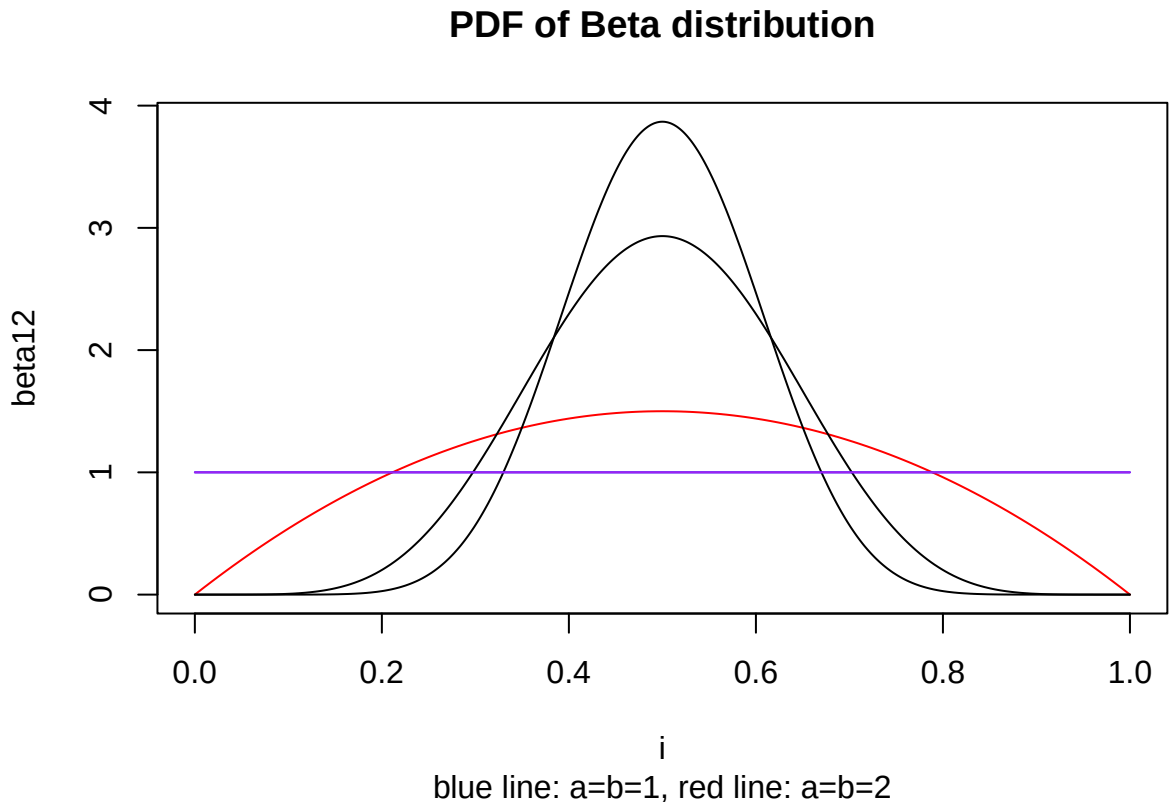
Smapling from the Beta distribution

This is the pdf of the Beta distribution.

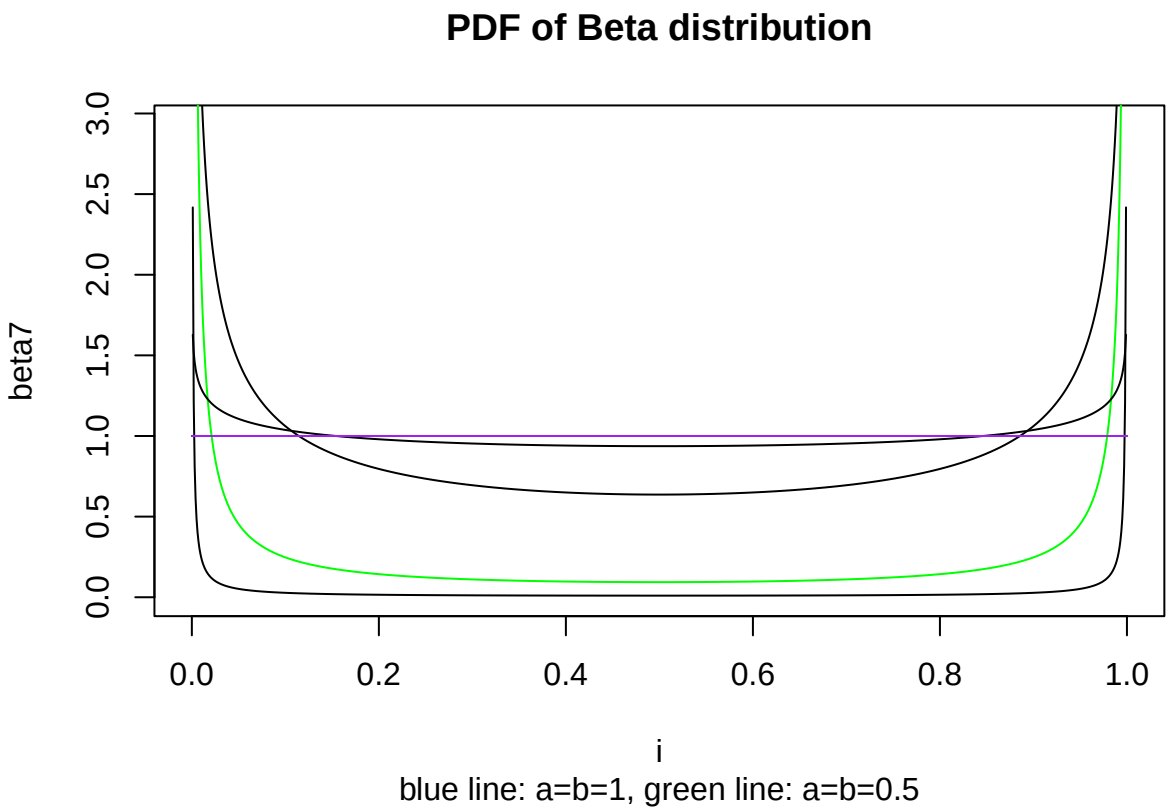
$$f(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

```
n <- 1000
i <- seq(0, 1, 1/n)

beta1 <- dbeta(i, 1,1)
beta2 <- dbeta(i, 2,2)
beta7 <- dbeta(i, 7,7)
beta12 <- dbeta(i, 12,12)
beta.5 <- dbeta(i, .5, .5)
beta.005 <- dbeta(i, .005, .005)
beta.05 <- dbeta(i, .05, .05)
beta.05 <- dbeta(i, .05, .05)
beta.9 <- dbeta(i, .9, .9)

norm <- dnorm(i)
uni <- dunif(i)
```

```r
plot(i, beta12, type="n", main="PDF of Beta distribution",
     sub="blue line: a=b=1, red line: a=b=2")
lines(i, beta2, col="red")
lines(i, beta12)
lines(i, beta7)
lines(i, beta1, col="blue")
lines(i, uni, col="purple")
```




```r
# plot(seq(0,20), seq(0,4, 4/20), type="n")
plot(i, beta7, type="n", main="PDF of Beta distribution",
     sub="blue line: a=b=1, green line: a=b=0.5")
# lines(beta2)
# lines(beta7)
lines(i, beta1, col="blue")
lines(i, beta.5)
lines(i, beta.005)
lines(i, beta.05, col="green")
lines(i, beta.9)
lines(i, uni, col="purple")
```

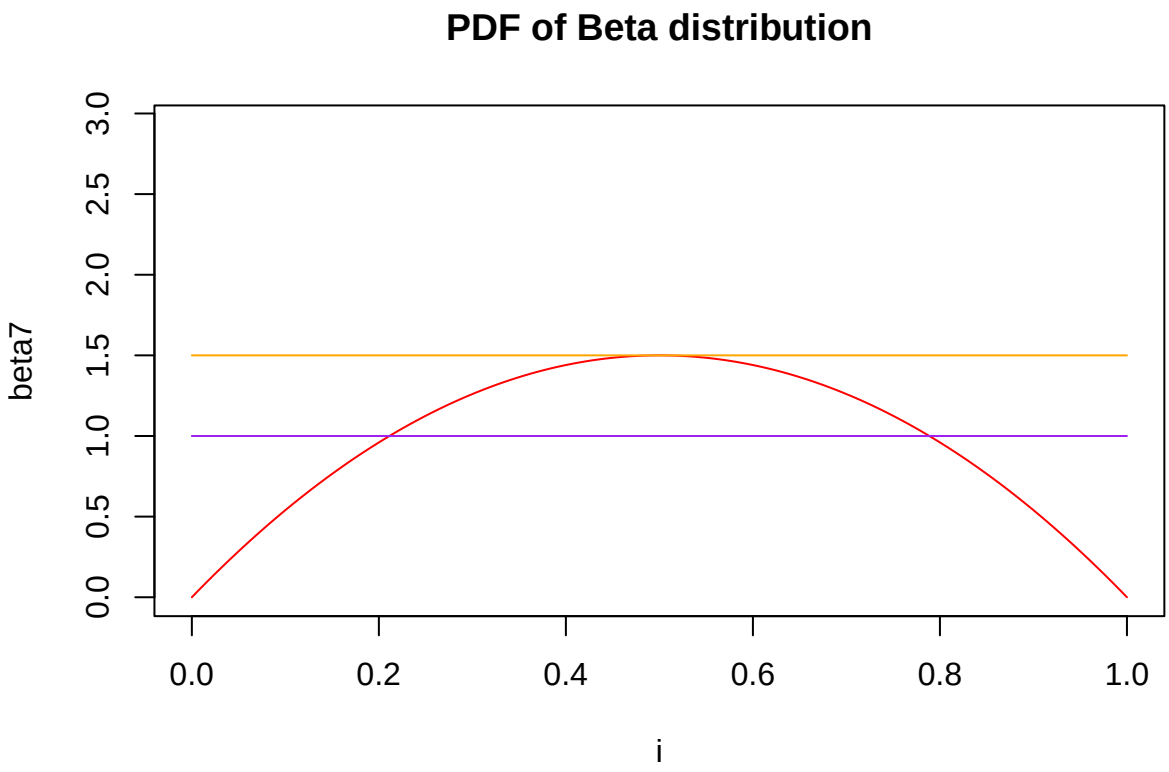



```
# lines(beta.05)
```

When observing the pdf of the Beta distributions with alpha and beta values larger than 1, the curves are upwards-bell-shaped, converging to 0 on both ends. An obvious choice of proposal distribution for the acceptance-rejection method is the uniform distribution. If values a and b are greater than 1, a straight line above the bell curve would allow us to propose samples that may well be samples from our target distribution, even if that may be inefficient for lower values.

Let $g(x)$ be the uniform density and $f(x)$ be the density of the Beta distribution. We then need to find the constant c, where $\forall x \in [0,1] : c * g(x) > f(x)$. When looking at the red line in the PDF of the Beta distribution, the curve peaks at around 1.5. We can confirm this by computing the density at the median (0.5)= $d_beta (0.5, 2, 2)= 1.5. The density of the uniform distribution is always 1, so a constant c=1.5$ would be sufficient here, as shown in the following plot:

```
plot(i, beta7, type="n", main="PDF of Beta distribution",
     sub="red line: f(x), purple line: g(x), orange line: g(x)*1.5")
lines(i, beta2, col="red")
lines(i, uni, col="purple")
lines(i, uni*1.5, col="orange")
```



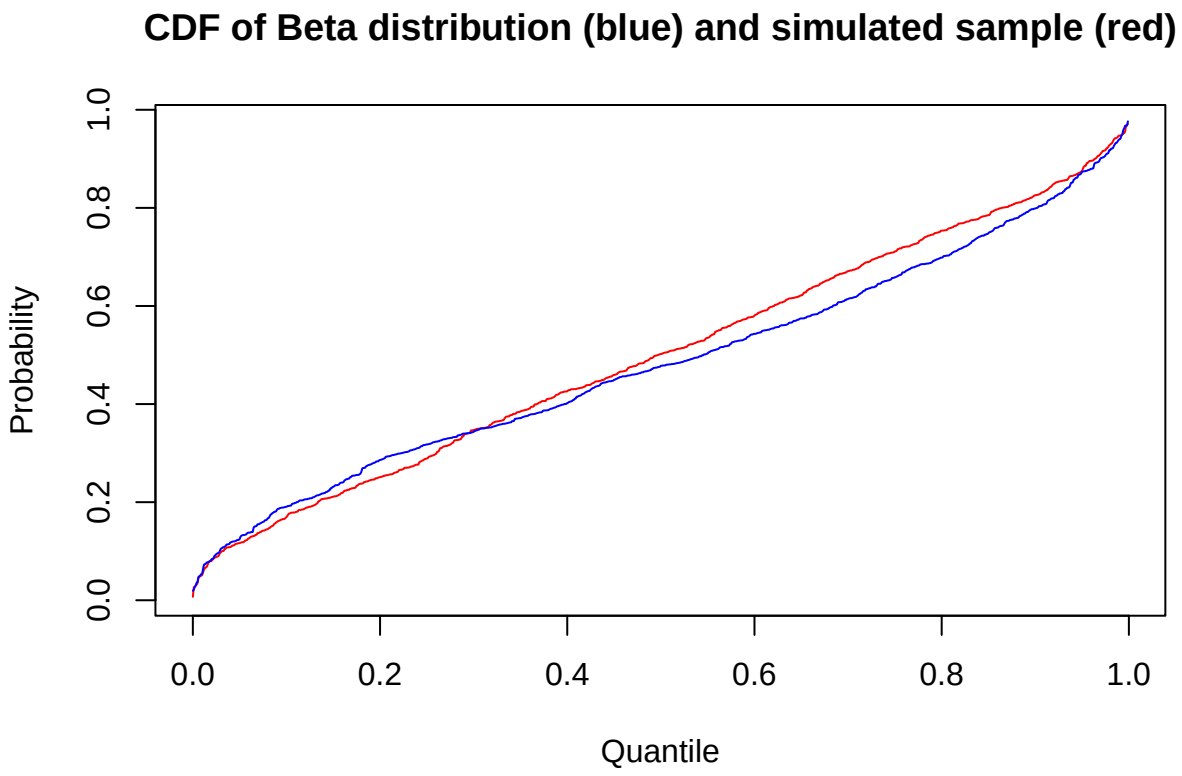
red line: f(x), purple line: g(x), orange line: g(x)*1.5

```r
accept_reject_beta <- function(n, a=2, b=2, c=1.5, get=T, plot=T){
  n_accepted <- 0
  n_rejected <- 0
  x <- numeric(n)

  while (n_accepted<n){
    u <- runif(1)
    # get a proposal sample
    y <- runif(1)
    # test the proposed sample
    if (u <= dbeta(y, a, b) / c){
      n_accepted <- n_accepted + 1
      x[n_accepted] <- y
    } else {
      n_rejected <- n_rejected + 1
    }
  }
  print(paste("Sampling complete -- Rejection proportion:", (n_rejected / (n+n_rejected)) %>% round(4),
      # plot the data
  if (plot){
    i <- seq(0, 1, 1/n)[1:n]
    plot(i, sort(x), type="n", xlab="Quantile", ylab="Probability",
         main="CDF of Beta distribution (blue) and simulated sample (red)")
    lines(i, sort(x), col="red")
    lines(i, rbeta(n, a, b) %>% sort(), col="blue")
  }
```

```
}
```

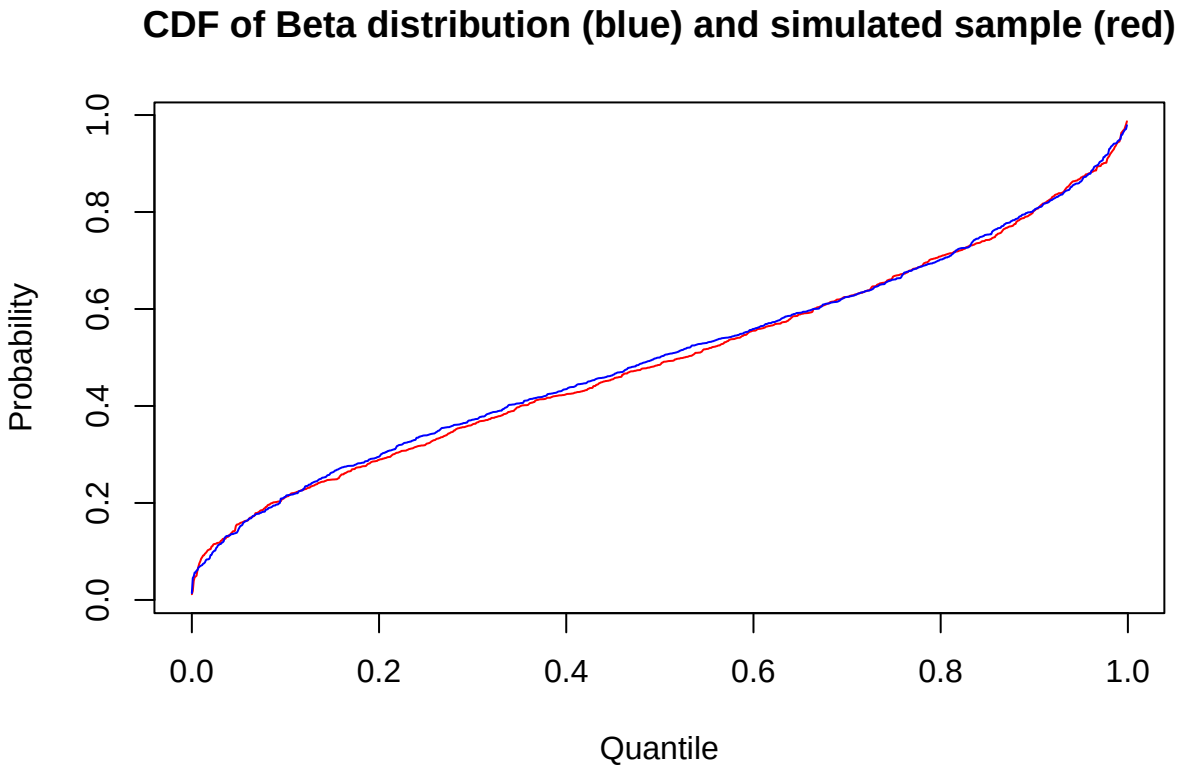
```
accept_reject_beta(n=n, a=2, b=2, c=1, get=F)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.1903 -- Iterations: 1235"
```



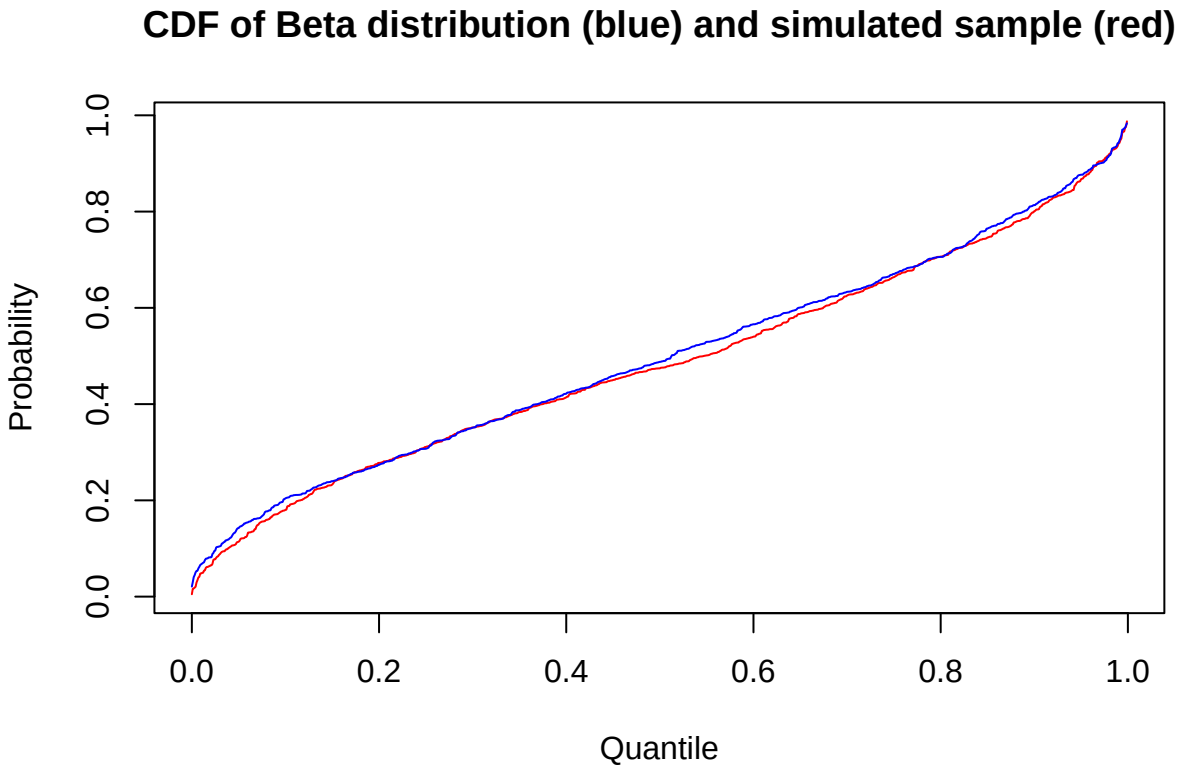

```
accept_reject_beta(n=n, a=2, b=2, c=1.5, get=F)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.3382 -- Iterations: 1511"
```



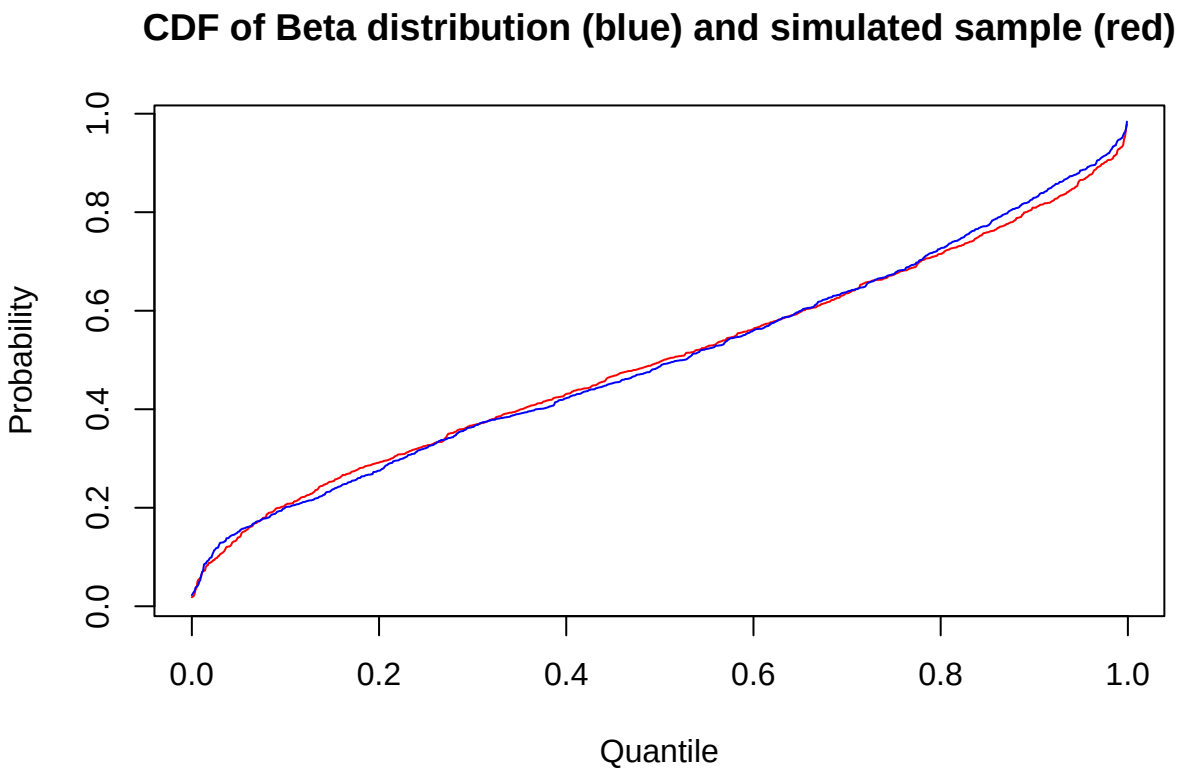

```
accept_reject_beta(n=n, a=2, b=2, c=3, get=F)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.6844 -- Iterations: 3169"
```




```
accept_reject_beta(n=n, a=2, b=2, c=30, get=F)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.9668 -- Iterations: 30111"
```



The function above uses a static constant c to apply the acceptance-rejection method to gather samples from the Beta distribution using the uniform distribution to propose possible samples.
The test results show that the higher the constant c, i.e. the larger the distance between the PDFs, the more samples get rejected and the more iterations are required to gather the wanted number of samples. While using a value of $c = 1$ terminates the fastest and rejects the fewest samples, this option is not viable, because this way the requirement of $\forall x \in [0, 1] : c * g(x) > f(x)$ is no longer met.

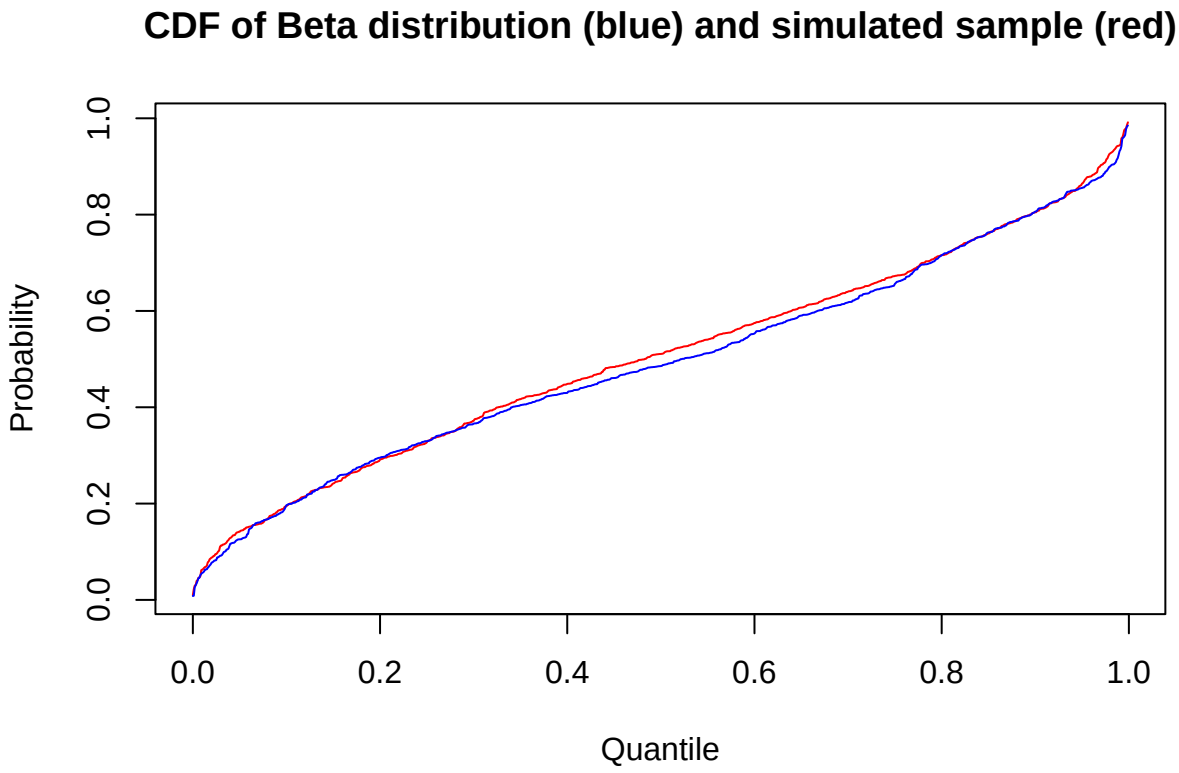
The plotted lines on the CDFs show that the simulated sample matches the target distribution quite closely.

Different values of a and b; a,b>=1

```
accept_reject_beta_autoc <- function(n, a=2, b=2, get=T, plot=T){
  n_accepted <- 0
  n_rejected <- 0
  x <- numeric(n)

  # compute the optimal value c
  # to avoid having to differentiate the pdf, i do a heuristic approach
  c <- dbeta(i, 4, 4) %>% max()

  while (n_accepted<n){
    u <- runif(1)
    # get a proposal sample
    y <- runif(1)
    # test the proposed sample
```

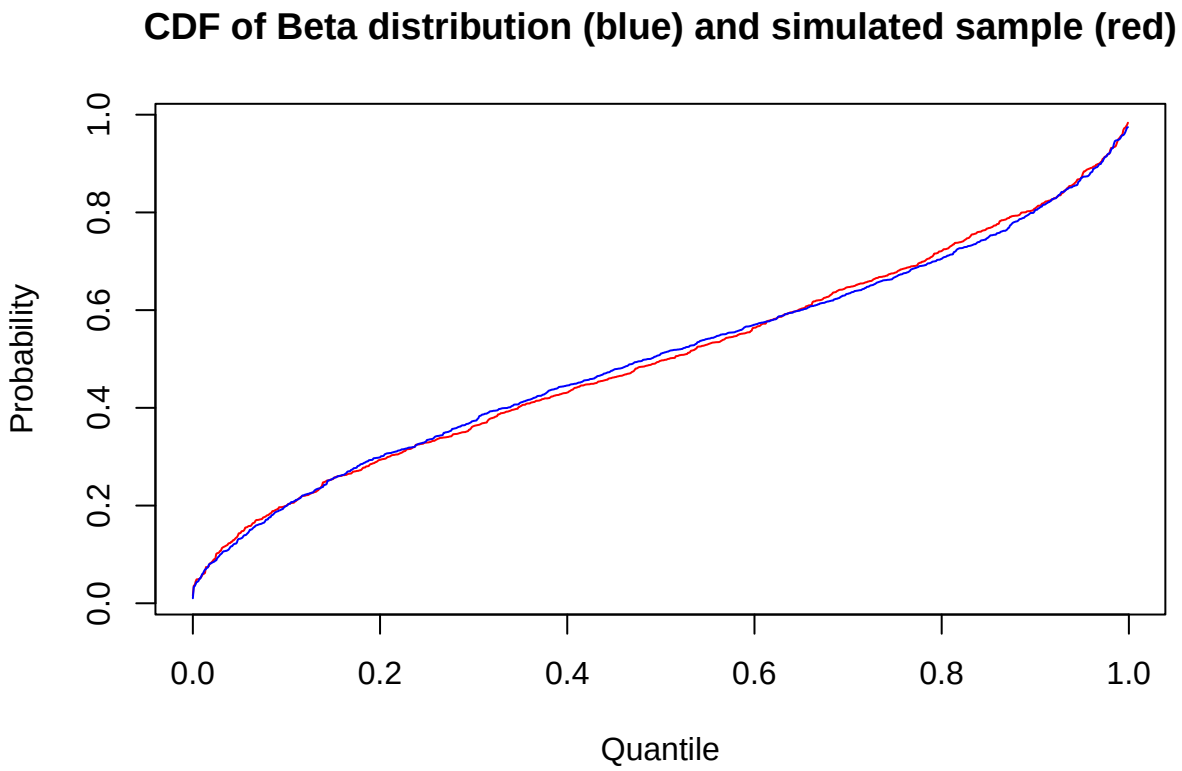
```
    if (u <= dbeta(y, a, b) / c){
      n_accepted <- n_accepted + 1
      x[n_accepted] <- y
    } else {
      n_rejected <- n_rejected + 1
    }
  }
  print(paste("Sampling complete -- Rejection proportion:", (n_rejected / (n+n_rejected)) %>% round(4),

    # plot the data
  if (plot){
    i <- seq(0, 1, 1/n)[1:n]
    plot(i, sort(x), type="n", xlab="Quantile", ylab="Probability",
         main="CDF of Beta distribution (blue) and simulated sample (red)")
    lines(i, sort(x), col="red")
    lines(i, rbeta(n, a, b) %>% sort(), col="blue")
  }

}

accept_reject_beta_autoc(n=n, a=2, b=2, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.5355 -- Iterations: 2153"
```



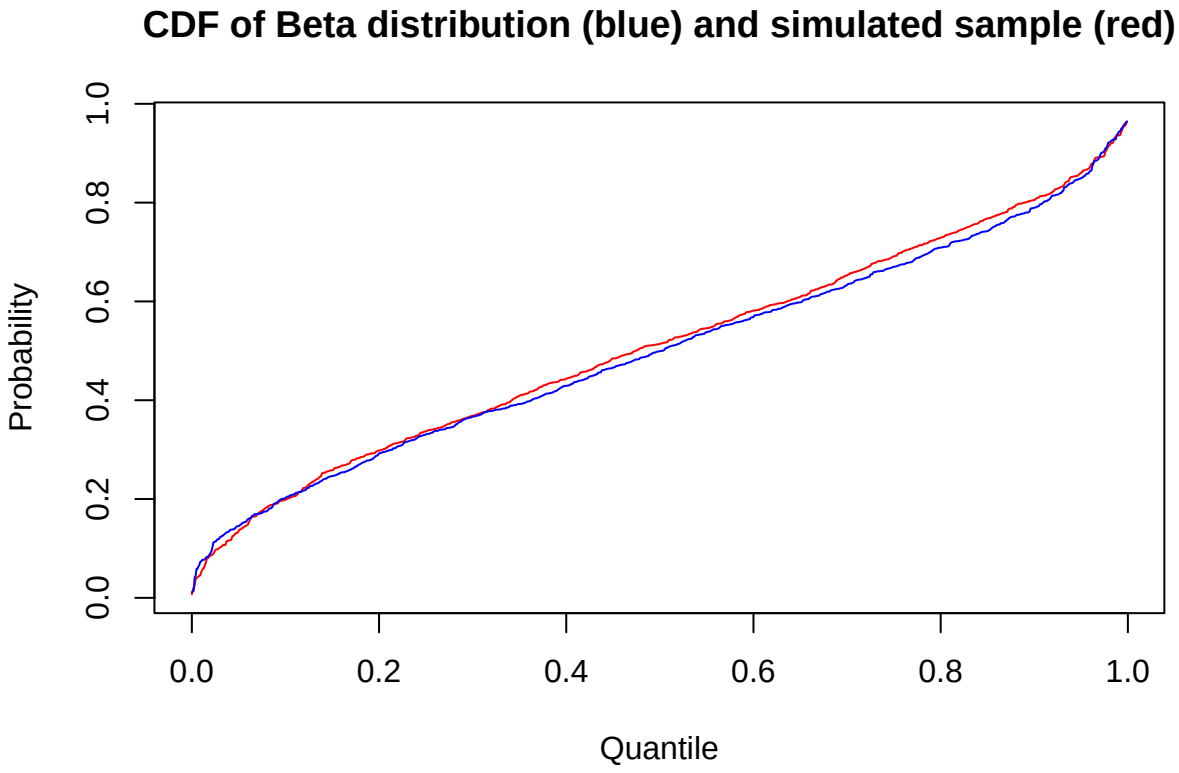
```
accept_reject_beta_autoc(n=n, a=2, b=2, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.5538 -- Iterations: 2241"
```



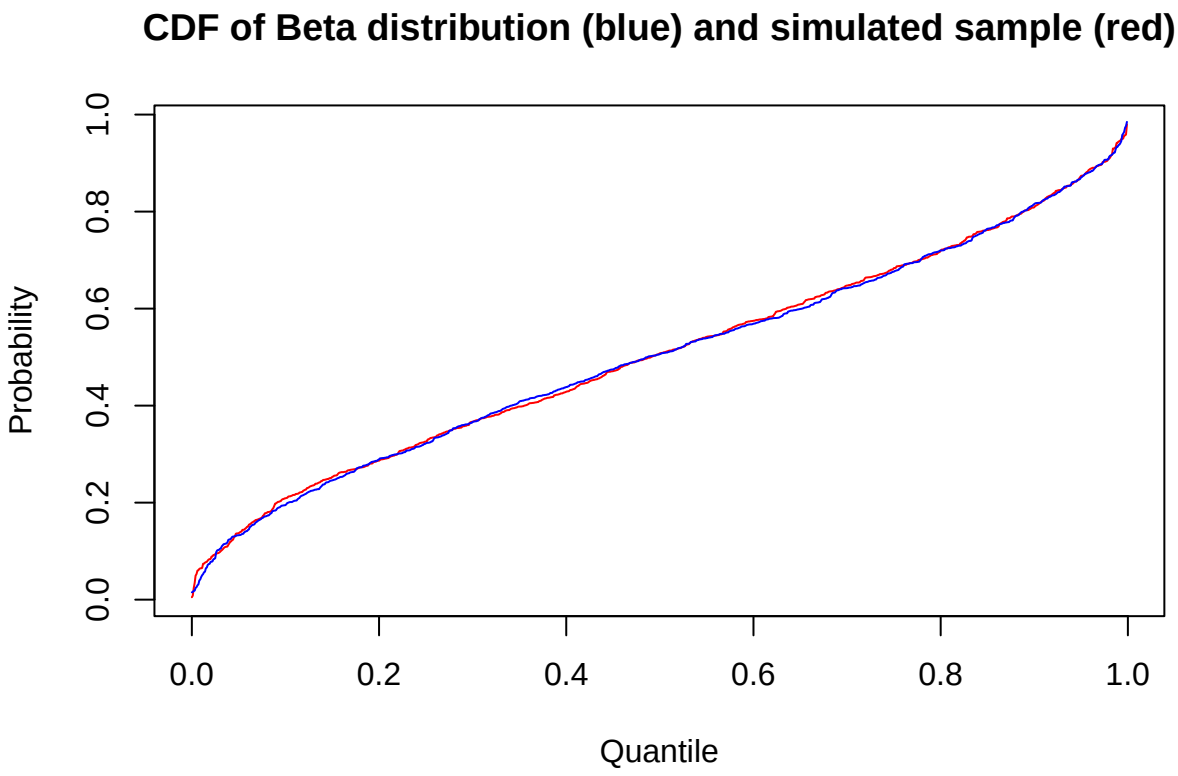

```
accept_reject_beta_autoc(n=n, a=2, b=2, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.5495 -- Iterations: 2220"
```




```
accept_reject_beta_autoc(n=n, a=2, b=2, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.5614 -- Iterations: 2280"
```



The approach above uses a heuristic approach of calculating 1000 samples from the pdf of beta and grabbing the max value. While this approach is approximately doing what we want, we still cannot guarantee every value being the requirement defined before to be always true.

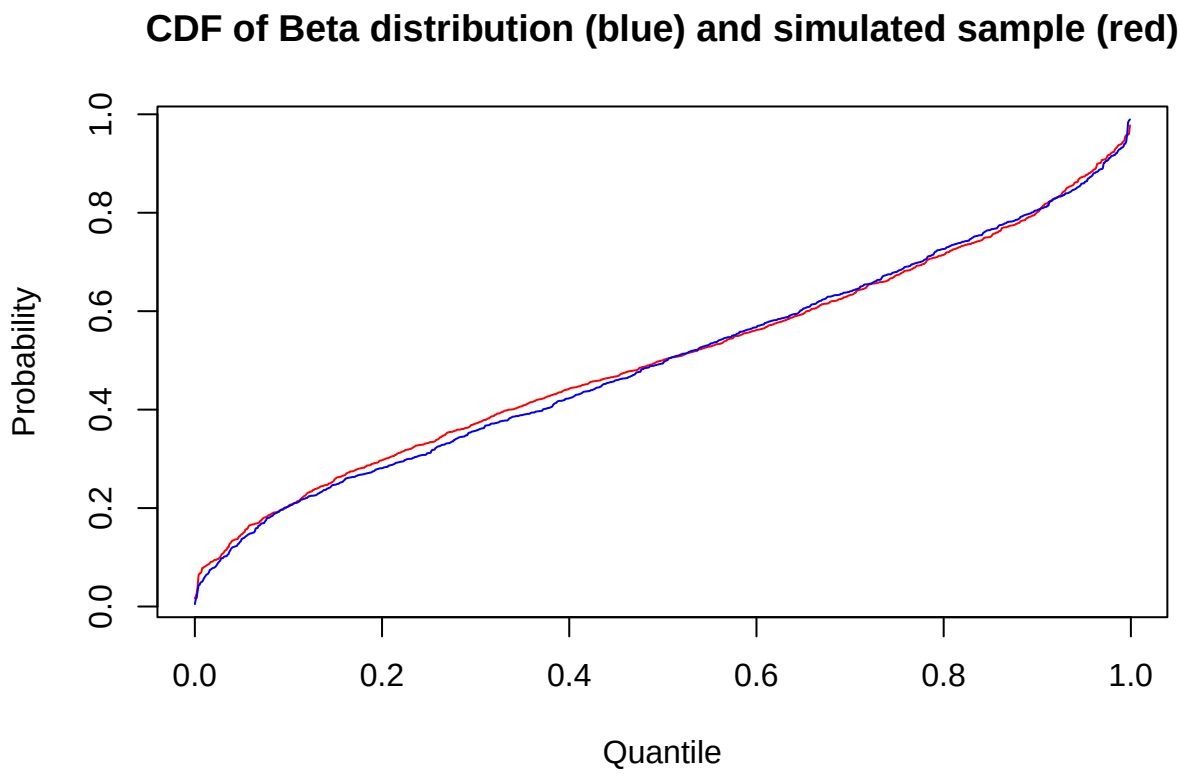
Again, the cdf plots show

The maximum of the Beta pdf is generally located at $x = (\alpha - 1)/(\alpha + \beta - 2)$, which allows us to make an more exact determination of a good c value:

```
accept_reject_beta_autoc <- function(n, a=2, b=2, get=T, plot=F){
  n_accepted <- 0
  n_rejected <- 0
  x <- numeric(n)

  # compute the optimal value c
  # to avoid having to differentiate the pdf, i do a heuristic approach
  quantile_of_max_density_beta <- (a-1)/(a+b-2)
  c <- dbeta(quantile_of_max_density_beta, a, b)

  while (n_accepted<n){
    u <- runif(1)
    # get a proposal sample
    y <- runif(1)
    # test the proposed sample
    if (u <= dbeta(y, a, b) / c){
      n_accepted <- n_accepted + 1
      x[n_accepted] <- y
    } else {
```

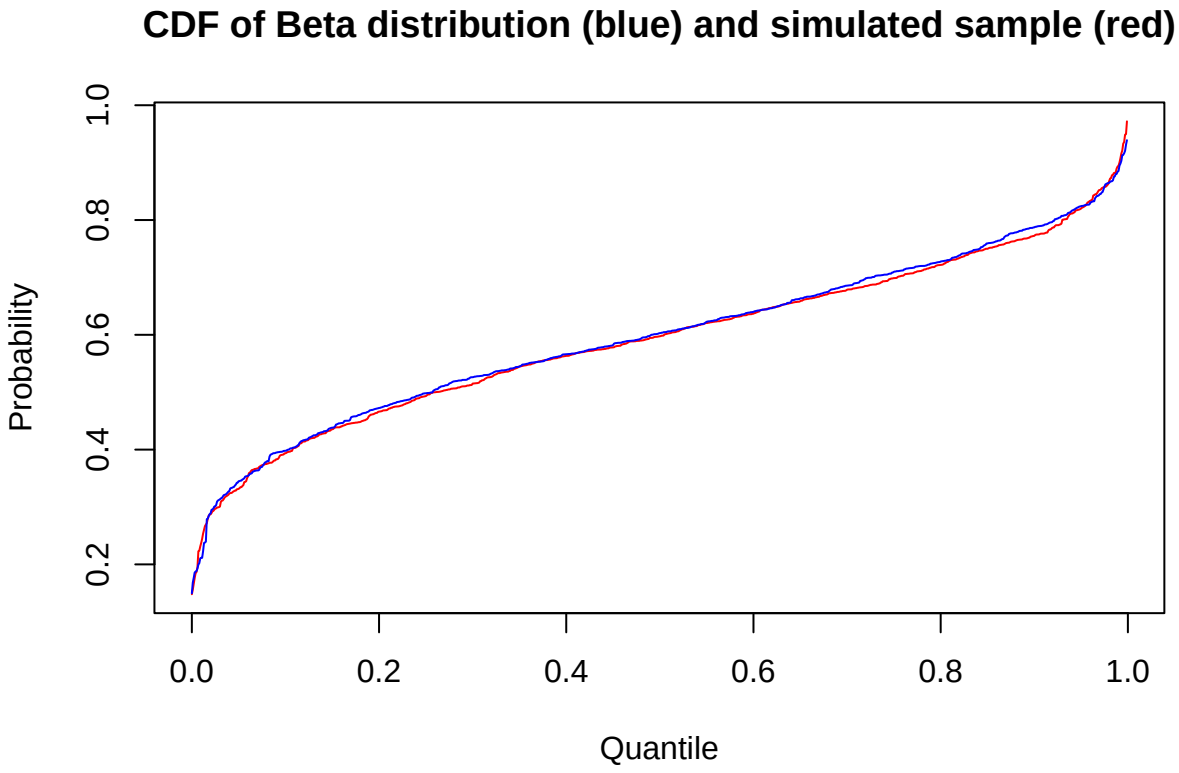
```
      n_rejected <- n_rejected + 1
    }
  }
  print(paste("Sampling complete -- Rejection proportion:", (n_rejected / (n+n_rejected)) %>% round(4),

  # plot the data
  if (plot){
    i <- seq(0, 1, 1/n)[1:n]
    plot(i, sort(x), type="n", xlab="Quantile", ylab="Probability",
         main="CDF of Beta distribution (blue) and simulated sample (red)")
    lines(i, sort(x), col="red")
    lines(i, rbeta(n, a, b) %>% sort(), col="blue")
  }
}

accept_reject_beta_autoc(n=n, a=2, b=2, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.3464 -- Iterations: 1530"
```



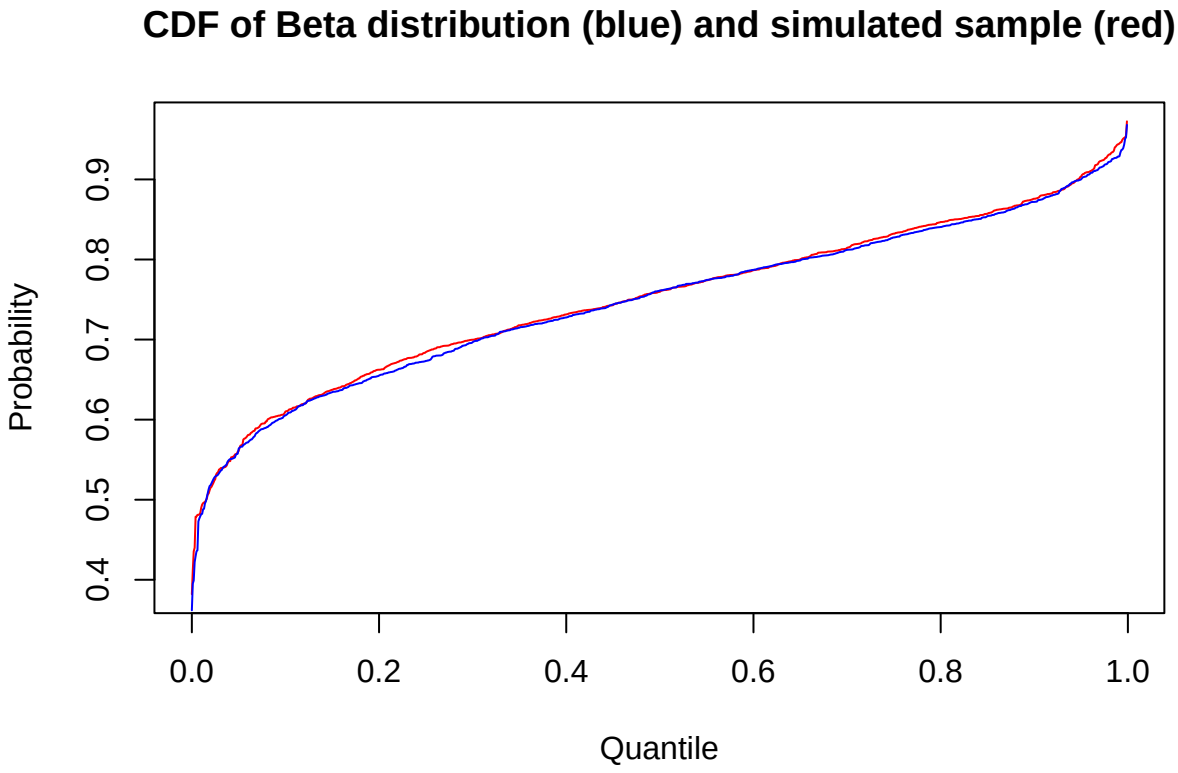

```
accept_reject_beta_autoc(n=n, a=6, b=4, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.6104 -- Iterations: 2567"
```



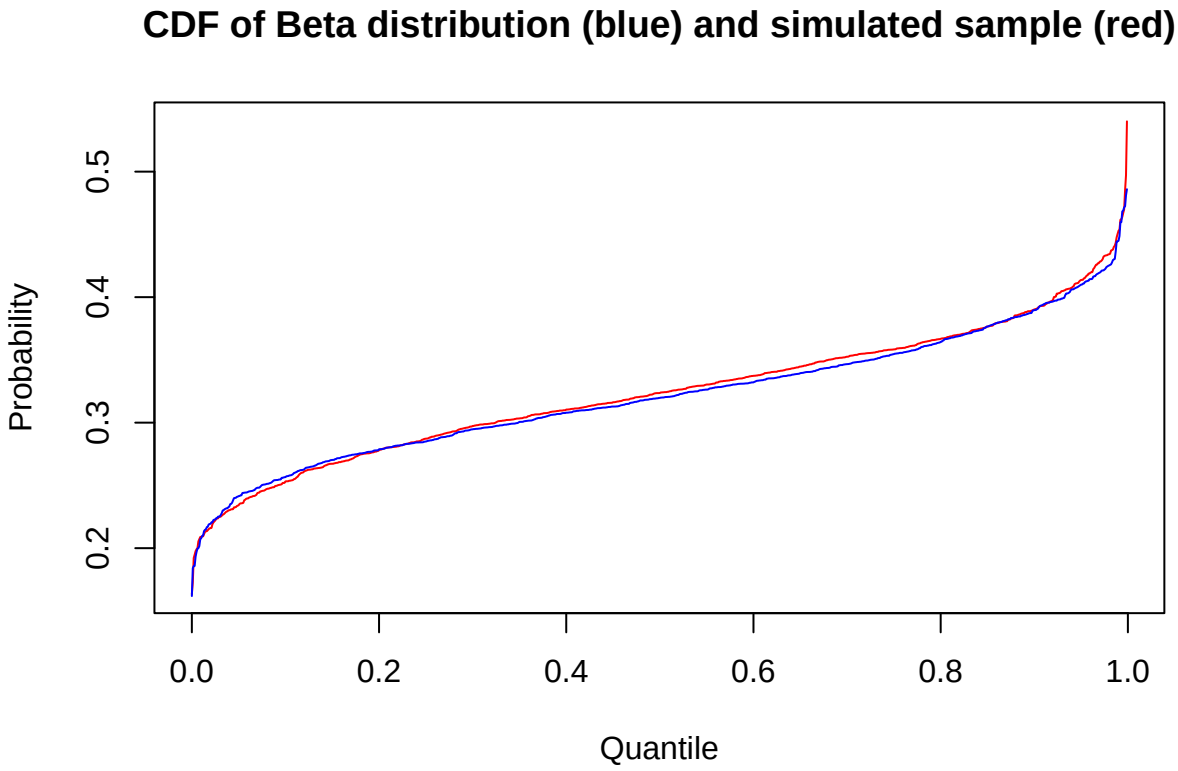

```
accept_reject_beta_autoc(n=n, a=12, b=4, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.7366 -- Iterations: 3797"
```




```
accept_reject_beta_autoc(n=n, a=25, b=53, get=F, plot=T)
```

```
## [1] "Sampling complete -- Rejection proportion: 0.8573 -- Iterations: 7006"
```



This approach is mathematically more correct, even though the required iterations and rejection proportion are pretty much the same as when using the heuristic approach. On the other hand, this approach is more efficient overall, as the function saves time by not having to generate a whole extra sample just to get a good maximum value.

Here the cdf plot lines still show promising matches of the simulated sample distribution to the target distribution.